621 HW4

May 8, 2019

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

1 Porblem 1. Comparing different Monte Carlo schemes

1.1

In this problem, we will construct a scheme to implement the Monte Carlo method in order to price the options and record the standard error and time consumed of the doing this estimation.

First of all, we want to gather all the option's information into one type of class and then construct a Monte Carlo funciton only contains the main parameters "m" and "n", and the options' information is another input of this function as whole.

```
In [67]: import numpy as np
         from scipy.stats import norm
         import time
         import pandas as pd
         import matplotlib.pyplot as plt
         from mpl_toolkits.mplot3d import Axes3D

         class derivatives(object):
             def __init__(self,Strike,Maturity,Spot_p,r,q,vol,opt_type):
                 self.K = Strike
                 self.T = Maturity
                 self.S0 = Spot_p
                 self.r = r
                 self.q = q
                 self.vol = vol
                 self.opt_type = opt_type


         der1 = derivatives(Strike=100,Maturity = 1,Spot_p = 100,r = 0.06,
                            q = 0.03,vol = 0.2,opt_type = "call")
         der2 = derivatives(Strike=100,Maturity = 1,Spot_p = 100,r = 0.06,
                            q = 0.03,vol = 0.2,opt_type = "put")
```

After cunstructing this option class, we define the Monte Carlo function.

```
In [3]: def Monte_Carlo1(n_,m_,der):
            start = time.time()
            dt = der.T/n_
            nudt = (der.r-der.q-(der.vol)**2*0.5)*dt
            sigdt = der.vol*np.sqrt(dt)
            dis = np.exp(-der.r*der.T)
            sum_C = 0
            sum_C2 = 0
            for i in range(int(m_)):
                lnSt = np.log(der.S0)
                for j in range(int(n_)):
                    z = np.random.randn()
                    lnSt += nudt+sigdt*z
                if der.opt_type == "call":
                    C = dis*max(np.exp(lnSt)-der.K,0)
                else:
                    C = dis*max(-np.exp(lnSt)+der.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, se, (end-start)
```

Now we can implement the Monte Carlo scheme on the call and put option pricing. Here we get the n=300, 400 and m = 1 million and 2 million. We only give several time consuming and error results of call options, for the put options we only give the estimated prices.

```
In [4]: result_table1 = pd.DataFrame(np.zeros((2,2)))
        result_table2 = pd.DataFrame(np.zeros((2,2)))
        result_table3 = pd.DataFrame(np.zeros((2,2)))
        for i in range(1,3):
            for j in range(1,3):
                n = i*100+200
                m = j*1e6
                res = Monte_Carlo1(n_ = n,m_ = m,der = der1)
                result_table1.iloc[i-1,j-1] = res[0]
                result_table2.iloc[i-1,j-1] = res[1]
                result_table3.iloc[i-1,j-1] = res[2]
```

```
In [101]: result_table1.columns = ["m = 1 million","m = 2 million"]
          result_table1.index = ["n = 300","n = 400"]
          print(result_table1)

          m = 1 million  m = 2 million
n = 300       9.155295       9.142964
n = 400       9.157449       9.144916
```

The table above is the price estimated.

```
In [102]: result_table2.columns = ["m = 1 million","m = 2 million"]
          result_table2.index = ["n = 300","n = 400"]
          print(result_table2)

          m = 1 million  m = 2 million
n = 300       0.013717       0.009683
n = 400       0.013707       0.009703
```

The table above is the standard error of each estimation.

$$standard\ error = \frac{\sigma}{\sqrt{t}}$$

when n or m increase, the standard error turn to be going down.

```
In [103]: result_table3.columns = ["m = 1 million","m = 2 million"]
          result_table3.index = ["n = 300","n = 400"]
          print(result_table3)

          m = 1 million  m = 2 million
n = 300     173.728559     337.100816
n = 400     222.848288     442.305521
```

The table above is the time consumed for each routine. It is intuitive that when n and m increase, the time consuming is increasing too.

```
In [8]: rsult_table4 = pd.DataFrame([Monte_Carlo1(n_ = 300,m_ = 1e6,der = der2)[0],
                                     Monte_Carlo1(n_ = 400,m_ = 1e6,der = der2)[0],
                                     Monte_Carlo1(n_ = 400,m_ = 2e6,der = der2)[0]])

In [105]: rsult_table4.columns = ["Price"]
          rsult_table4.index = ["n = 300,m = 1 million",
                                "n = 400,m = 1 million",
                                "n = 400,m = 2 million"]
          print(rsult_table4)

                          Price
n = 300,m = 1 million  6.263642
n = 400,m = 1 million  6.256176
n = 400,m = 2 million  6.272521
```

The above is the estimatd price for put options.

1.2

In this problem, we should use several control variate method. In order to compare each other, we turn the simulation number to 10000 to get the results more quickly. First of all, we compute the normal Monte Carlo method results for put and call options.

```
In [117]: res_call1 = Monte_Carlo1(n_ = 300,m_ = 10000,der = der1)
          res_put1 = Monte_Carlo1(n_ = 300,m_ = 10000,der = der2)
```

Now,we construct a function to implement the antithetic variates method, which will generate a pair of random variables that have the mean value of what we want.

```
In [115]: def Monte_Carlo_Anti(n_,m_,der):
              start = time.time()
              dt = der.T/n_
              nudt = (der.r-der.q-(der.vol)**2*0.5)*dt
              sigdt = der.vol*np.sqrt(dt)
              dis = np.exp(-der.r*der.T)
              sum_C = 0
              sum_C2 = 0
              for i in range(int(m_)):
                  lnSt1 = np.log(der.S0)
                  lnSt2 = np.log(der.S0)
                  for j in range(int(n_)):
                      z = np.random.randn()
                      lnSt1 += nudt+sigdt*z
                      lnSt2 += nudt-sigdt*z
                  if der.opt_type == "call":
                      C = dis*(max(np.exp(lnSt1)-der.K,0)+max(np.exp(lnSt2)-der.K,0))/2
                  else:
                      C = dis*(max(-np.exp(lnSt1)+der.K,0)+max(-np.exp(lnSt2)+der.K,0))/2
                  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, se, (end-start)

In [116]: res_call2 = Monte_Carlo_Anti(n_=300,m_=10000,der=der1)
          res_put2 = Monte_Carlo_Anti(n_=300,m_=10000,der=der2)
```

Then we contruct the delta-based variate control method, which relys on the delta hedging method. Here we need to first contrust the delta function that return the right delta for a given stock price and option price and then we define the main function that implement the delta-based variate control method.

```
In [51]: def Black_Scholes_delta(St,t,K,T,sig,r,div):
             d1 = (np.log(St/K)+(r+sig**2/2)*(T-t))/(sig*np.sqrt(T-t))
             return np.exp(-div*(T-t))*norm.cdf(d1)
```

```
        def Monte_Carlo_Del(n_,m_,der):
            start = time.time()
            dt = der.T/n_
            nudt = (der.r-der.q-(der.vol)**2*0.5)*dt
            sigdt = der.vol*np.sqrt(dt)
            dis = np.exp(-der.r*der.T)
            erddt = np.exp((der.r-der.q)*dt)
            beta1 = -1

            sum_C = 0
            sum_C2 = 0
            for i in range(int(m_)):
                St = der.S0
                cv = 0
                cv2 = 0
                for j in range(int(n_)):
                    t = (j)*dt
                    delta = Black_Scholes_delta(St = St,t=t,K=der.K,T=der.T,sig=der.vol,r=der
                    z = np.random.randn()
                    Stn = St*np.exp(nudt+sigdt*z)
                    cv += delta*(Stn-St*erddt)*np.exp(der.r*(der.T-t-dt))
                    cv2 += (delta-1)*(Stn-St*erddt)*np.exp(der.r*(der.T-t-dt))
                    St = Stn
                if der.opt_type == "call":
                    C = dis*(max(St-der.K,0)+beta1*cv)
                else:
                    C = dis*(max(-St+der.K,0)+beta1*cv2)
                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, se, (end-start)

In [52]: res_call3 = Monte_Carlo_Del(n_=300,m_=10000,der=der1)
         res_put3 = Monte_Carlo_Del(n_=300,m_=10000,der=der2)
```

Then we combine the two methods into one.

```
In [45]: def Monte_Carlo_Anti_Del(n_,m_,der):
             start = time.time()
             dt = der.T/n_
             nudt = (der.r-der.q-(der.vol)**2*0.5)*dt
             sigdt = der.vol*np.sqrt(dt)
             dis = np.exp(-der.r*der.T)
             erddt = np.exp((der.r-der.q)*dt)
             beta1 = -1
             sum_C = 0
```

```
            sum_C2 = 0
            for i in range(int(m_)):
                St1 = der.S0
                St2 = der.S0
                cv1 = 0
                cv2 = 0
                cv3 = 0
                cv4 = 0
                for j in range(int(n_)):
                    t = (j)*dt
                    delta1 = Black_Scholes_delta(St = St1,t=t,K=der.K,T=der.T,sig=der.vol,r=de
                    delta2 = Black_Scholes_delta(St = St2,t=t,K=der.K,T=der.T,sig=der.vol,r=de
                    z = np.random.randn()
                    Stn1 = St1*np.exp(nudt+sigdt*z)
                    Stn2 = St2*np.exp(nudt+sigdt*(-z))
                    cv1 += delta1*(Stn1-St1*erddt)*np.exp(der.r*(der.T-t-dt))
                    cv2 += delta2*(Stn2-St2*erddt)*np.exp(der.r*(der.T-t-dt))
                    cv3 += (delta1-1)*(Stn1-St1*erddt)*np.exp(der.r*(der.T-t-dt))
                    cv4 += (delta1-1)*(Stn2-St2*erddt)*np.exp(der.r*(der.T-t-dt))
                    St1 = Stn1
                    St2 = Stn2
                if der.opt_type == "call":
                    C = 0.5*dis*((max(St1-der.K,0)+beta1*cv1)+
                             (max(St2-der.K,0)+beta1*cv2))
                else:
                    C = 0.5*dis*((max(-St1+der.K,0)+beta1*cv3)+
                             (max(-St2+der.K,0)+beta1*cv4))
                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, se, (end-start)

In [46]: res_call4 = Monte_Carlo_Anti_Del(n_=300,m_=10000,der=der1)
         res_put4 = Monte_Carlo_Anti_Del(n_=300,m_=10000,der=der2)
```

Now, we compile the reults into one table and find some conclusion.

```
In [118]: result_table5 = pd.DataFrame([[res_call1[0],res_call1[1],res_call1[2]],
                                        [res_call2[0],res_call2[1],res_call2[2]],
                                        [res_call3[0],res_call3[1],res_call3[2]],
                                        [res_call4[0],res_call4[1],res_call4[2]]],
                                       index = ["MC","Antithetic","Delta","Combined"],
                                       columns = ["Price","RMSE","Time consumed"])

          result_table6 = pd.DataFrame([[res_put1[0],res_put1[1],res_put1[2]],
                                        [res_put2[0],res_put2[1],res_put2[2]],
```

```
                                        [res_put3[0],res_put3[1],res_put3[2]],
                                        [res_put4[0],res_put4[1],res_put4[2]]],
                                    index = ["MC","Antithetic","Delta","Combined"],
                                    columns = ["Price","RMSE","Time consumed"])
print(result_table5)
print(result_table6)
```

```
               Price      RMSE  Time consumed
MC          9.126826  0.135405       2.078444
Antithetic  9.113590  0.071916       3.055828
Delta       9.144749  0.007589     210.310737
Combined    9.135663  0.003633     412.011618
               Price      RMSE  Time consumed
MC          6.281213  0.092091       2.067505
Antithetic  6.253703  0.046381       3.082759
Delta       6.270389  0.005207     209.599672
Combined    6.360943  0.060237     411.252035
```

The first table is for call options and the second is for put options. As we can see, for the call options, the price is more accurarte when using the antithetic and delta-based control variate method combined. The RMSE is much smaller when using the variate control methods. For the time consumed, the antithetic way only use 3 seconds to do the same amount of loops and give a fairly well results. However for call options the combined method is the best one.

For put options, the combined method get a large RMSE and the price is not very accurate as we expected. The Delta based variate control method may be more proper to use in prcing the put options.

2 Problem 2. Multiple Monte Carle Processes

2.1

First we compute the numebr of shares and the amount in the Yuan that the portfolio cantains when it is started.

```
In [56]: nx = 1e7*0.4/80
         ny = 1e7*0.3/90000
         yuanz = 1e7*0.3*6.1
```

2.2

Now we compute the VaR and CVaR in one function.

```
In [57]: def Monte_Carlo2(n_,m_,T,X0,Y0,Z0):
             dt = T/n_
             var_list = []
             for i in range(int(m_)):
                 Xt = X0
                 Yt = Y0
```

```
            Zt = Z0
            for j in range(int(n_)):
                t = j*dt
                z1 = np.random.randn()
                z2 = np.random.randn()
                z3 = np.random.randn()
                Xt += 0.01*Xt*dt+0.3*Xt*z1*np.sqrt(dt)
                Yt += 100*(90000+1000*t-Yt)*dt+np.sqrt(Yt)*z2*np.sqrt(dt)
                Zt += 5*(6-Zt)*Zt*dt+0.01*np.sqrt(Zt)*z3*np.sqrt(dt)
            P = nx*Xt+ny*Yt+yuanz/Zt
            var_list.append(P)
        var_list = np.asarray(var_list)
        quantile = np.quantile(var_list,0.01)
        VaR = 1e7-quantile
        cVaR = np.mean(1e7-var_list[var_list<=quantile])
        return var_list, VaR, cVaR
```

```
In [58]: a1,a2,a3 = Monte_Carlo2(n_=np.ceil(10/252/0.001),
                                 m_=3e4,T=10/252,X0=80,
                                 Y0=90000,Z0=6.1)
```

For VaR, we need to find the exact quantile of the while price sequence.

$$VaR_\alpha(X) = -inf\{F_X(x) > \alpha\}$$

Note that what we use here is the losses' quantile, the VaR we computed is presented below.

```
In [110]: print(a2)
```

```
487428.3444896247
```

2.3

After computing the VaR, we give the result of CVaR. CVaR is the conditional VaR which is the conditional expectation of the tail of loss distribution.

$$CVaR_{1-\alpha}(X) = \frac{1}{\alpha}\int_0^\alpha VaR_{1-\gamma}(X)d\gamma$$

We just average the losses that larger than the VaR we get. below is the CVaR we computed.

```
In [111]: print(a3)
```

```
557456.5080192907
```

3 Problem 3. Generating correlated BM and pricing Basket options

3.1

We construct a function to do the Cholesky decomposition for a given mattrix A. We need to get the matrix satisfy the formula $A = LL^T$

$$L_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2} L_{i,j} = \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} \right) \quad \text{for } i > j$$

```
In [63]: 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
         A_ = np.matrix([[1,0.5,0.2],
                         [0.5,1,-0.4],
                         [0.2,-0.4,1]])
         print(Cholesky(A_))
```

```
[[ 1.          0.          0.        ]
 [ 0.5         0.8660254   0.        ]
 [ 0.2        -0.57735027  0.79162281]]
```

As we can see, the function decompose the matrix A into a lower and upper triangular matrix.

3.2

Now we simulate these three asset and plot the route of them.

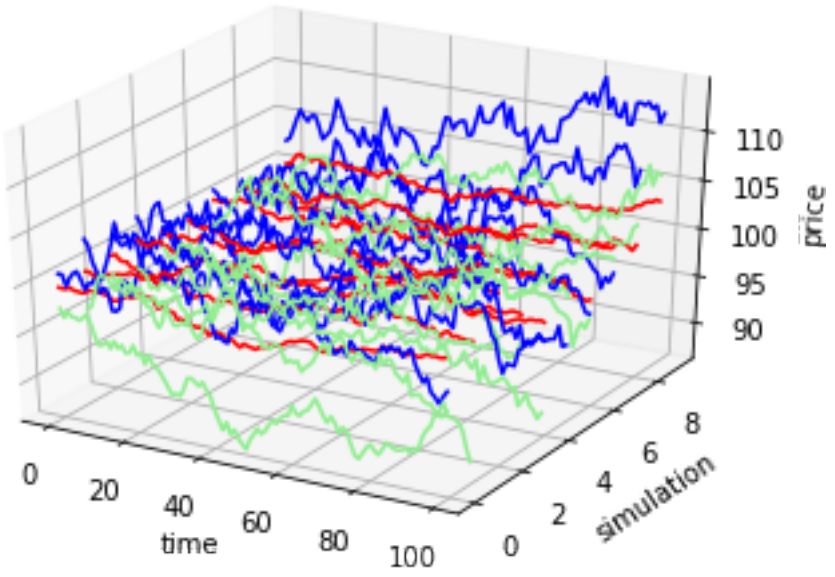
```
In [64]: def Monte_Carlo3(n_,m_,T,S0,mu0,sig0,A):
             dt = T/n_
             result = []
             L = Cholesky(A)
             for i in range(int(m_)):
                 Xt = S0[0]
                 Yt = S0[1]
                 Zt = S0[2]
                 res1 = []
                 for j in range(int(n_)):
```

```
                    row_z = np.random.randn(3)
                    z = np.dot(L,row_z)
                    z1 = z[0]
                    z2 = z[1]
                    z3 = z[2]
                    Xt += mu0[0]*Xt*dt+sig0[0]*Xt*z1*np.sqrt(dt)
                    Yt += mu0[1]*Yt*dt+sig0[1]*Yt*z2*np.sqrt(dt)
                    Zt += mu0[2]*Zt*dt+sig0[2]*Zt*z3*np.sqrt(dt)
                    res2 = [Xt,Yt,Zt]
                    res1.append(res2)
                result.append(res1)
            return result
```

```
In [65]: res3b = Monte_Carlo3(n_=100,m_=1000,T=100/365,
                              S0=[100,101,98],mu0=[0.03,0.06,0.02],
                              sig0=[0.05,0.2,0.15],A=A_)
```

```
In [73]: fig = plt.figure()
         ax = fig.add_subplot(111, projection='3d')
         for mm in range(10):
             xt = []
             yt = []
             zt = []
             for nn in range(len(res3b[mm])):
                 xt.append(res3b[mm][nn][0])
                 yt.append(res3b[mm][nn][1])
                 zt.append(res3b[mm][nn][2])
             ax.plot(range(100),np.asarray([mm]*100),xt,color = "red")
             ax.plot(range(100),np.asarray([mm]*100),yt,color = "blue")
             ax.plot(range(100),np.asarray([mm]*100),zt,color = "lightgreen")
         ax.set_xlabel('time')
         ax.set_ylabel('simulation')
         ax.set_zlabel('price')
```

```
Out[73]: Text(0.5,0,'price')
```



Here we use red line to denote the $S_1(t)$, blue line to denote the $S_2(t)$ and green line to denote $S_3(t)$. We only plot 10 simulations since it will be chaos when we plot all of 1000 routes.

3.3

In this question, we contruct a basket asset where each underlying asset has the wieght of $1/3$. Then we try to price an option on this basket asset. In this question we let the $T = 1$ and $n = 100$, $m = 10000$ to get the price.

```
In [74]: def Monte_Carlo4(n_,m_,T,S0,mu0,sig0,A,a,opt_type,K,r):
             dt = T/n_
             L = Cholesky(A)
             dis = np.exp(-r*T)
             sum_C = 0
             for i in range(int(m_)):
                 Xt = S0[0]
                 Yt = S0[1]
                 Zt = S0[2]
                 for j in range(int(n_)):
                     row_z = np.random.randn(3)
                     z = np.dot(L,row_z)
                     Xt = mu0[0]*Xt*dt+sig0[0]*Xt*z[0]*np.sqrt(dt)+Xt
                     Yt = mu0[1]*Yt*dt+sig0[1]*Yt*z[1]*np.sqrt(dt)+Yt
                     Zt = mu0[2]*Zt*dt+sig0[2]*Zt*z[2]*np.sqrt(dt)+Zt
                 Ut = a[0]*Xt+a[1]*Yt+a[2]*Zt
                 if opt_type == "call":
                     C = dis*max(Ut-K,0)
                 else:
```

```
                    C = dis*max(-Ut+K,0)
                sum_C += C
            result = sum_C/m_
            return result
```

```
In [85]: res3c_call = Monte_Carlo4(n_=100,m_=10000,T=1,S0 = [100,101,98],
                     mu0=[0.03,0.06,0.02],sig0=[0.05,0.2,0.15],A=A_,
                     a=[1/3,1/3,1/3],opt_type='call',K=100,r=0.06)
         res3c_put = Monte_Carlo4(n_=100,m_=10000,T=1,S0 = [100,101,98],
                     mu0=[0.03,0.06,0.02],sig0=[0.05,0.2,0.15],A=A_,
                     a=[1/3,1/3,1/3],opt_type='put',K=100,r=0.06)
```

```
In [84]: result_table3c = pd.DataFrame([res3c_call,res3c_put],
                                       index = ['call option','put option'],
                                       columns = ["price"])
         print(result_table3c)
```

```
                price
call option  4.785543
put option   1.719421
```

The simulated prices of basket options are above.

3.4

In this question, we price an option using the conditions described. In order to avoid the paradox, we give the condition the priority, the first condition have more priority than the second and the second have more than the third.

```
In [86]: def Monte_Carlo_all(n_,m_,T,S0,mu0,sig0,A,a,opt_type,K,r,B):
             dt = T/n_
             L = Cholesky(A)
             dis = np.exp(-r*T)
             sum_C = 0
             for i in range(int(m_)):
                 Xt = S0[0]
                 Yt = S0[1]
                 Zt = S0[2]
                 sum_Yt = Yt
                 sum_Zt = Zt
                 max_Yt = Yt
                 max_Zt = Zt
                 indicator = 0
                 for j in range(int(n_)):
                     row_z = np.random.randn(3)
                     z = np.dot(L,row_z)
                     Xt = mu0[0]*Xt*dt+sig0[0]*Xt*z[0]*np.sqrt(dt)+Xt
                     Yt = mu0[1]*Yt*dt+sig0[1]*Yt*z[1]*np.sqrt(dt)+Yt
```

```
                    Zt = mu0[2]*Zt*dt+sig0[2]*Zt*z[2]*np.sqrt(dt)+Zt
                    sum_Yt += Yt
                    sum_Zt += Zt
                    if Yt > max_Yt:
                        max_Yt = Yt
                    if Zt > max_Zt:
                        max_Zt = Zt
                    if Yt > B:
                        indicator = 1
                if indicator == 1:
                    C = dis*max(Yt-K,0)
                elif max_Yt>max_Zt:
                    C = dis*max(Yt**2-K,0)
                elif sum_Yt>sum_Zt:
                    C = dis*max(sum_Yt/(n_+1)-K,0)
                else:
                    Ut = a[0]*Xt+a[1]*Yt+a[2]*Zt
                    if opt_type == "call":
                        C = dis*max(Ut-K,0)
                    else:
                        C = dis*max(-Ut+K,0)
                sum_C += C
            result = sum_C/m_
            return result
```

```
In [87]: res3d = Monte_Carlo_all(n_=100,m_=100000,T=1,
                                  S0 = [100,101,98],mu0=[0.03,0.06,0.02],
                                  sig0=[0.05,0.2,0.15],A=A_,
                                  a=[1/3,1/3,1/3],opt_type='call',
                                  K=100,r=0.06,B=104)
```

```
In [88]: print(res3d)
```

```
102.46195587692827
```

The final call option price of this exotic option is what we estimated: about $102.

4 Bonus

4.1

For Heston Model, the process is described below:

$$dS_t = \mu S_t\,dt + \sqrt{v_t}S_t\,dW_t^S dv_t = \kappa(\theta - v_t)\,dt + \xi\sqrt{v_t}\,dW_t^v$$

In the paper, there are three function are implemented on the volatility process.

$\tilde{V}(t+\Delta t) = f_1(\tilde{V}(t)) - \kappa\Delta t(f_2(\tilde{V}(t)) - \theta) + \omega f_3(\tilde{V}(t))^\alpha \Delta W_V(t) V(t+\Delta t) = f_3(\tilde{V}(t+\Delta t))$

Where f_1 f_2 and f_3 are different functions. Now we contruct a Monte Carlo simulation to price the option whose underlying asset have such a process.

```
In [91]: def Monte_Carlo8(n_,m_,T,S0,V0,k,theta,sig,rho,r,K,f1,f2,f3):
             start = time.time()
             dt = T/n_
             dis = np.exp(-r*T)
             sum_C = 0
             sum_C2 = 0
             for i in range(int(m_)):
                 lnSt = np.log(S0)
                 Vtt = V0
                 Vt = V0
                 for j in range(int(n_)):
                     z1 = np.random.randn()
                     z2 = np.random.randn()
                     w1 = z1
                     w2 = rho*z1+np.sqrt(1-rho**2)*z2
                     Vtt = f1(Vtt)-k*dt*(f2(Vtt)-theta)+sig*f3(Vtt)**0.5*w1*np.sqrt(dt)
                     lnSt += (r-0.5*Vt)*dt+np.sqrt(Vt)*w2*np.sqrt(dt)
                     Vt = f3(Vtt)
                 St = np.exp(lnSt)
                 C = dis*max(St-K,0)
                 sum_C += C
                 sum_C2+= C**2
             mean_C = sum_C/m_
             bias = abs(6.8061-mean_C)
             se = np.sqrt((sum_C2-m_*mean_C**2)/(m_-1)/m_)
             end = time.time()
             return mean_C, bias, se, (end-start)

         def fa(x):
             return max(x,0)
         def fb(x):
             return abs(x)
         def fc(x):
             return x

In [94]: res_ab = Monte_Carlo8(n_=1000,m_=10000,T=1,S0=100,V0=0.010201,
                      k=6.21,theta=0.019,sig=0.61,rho=-0.7,r=0.0319,
                      K=100,f1=fa,f2=fa,f3=fa)

         res_re = Monte_Carlo8(n_=100,m_=10000,T=1,S0=100,V0=0.010201,
                      k=6.21,theta=0.019,sig=0.61,rho=-0.7,r=0.0319,
                      K=100,f1=fb,f2=fb,f3=fb)

         res_hm = Monte_Carlo8(n_=100,m_=10000,T=1,S0=100,V0=0.010201,
                      k=6.21,theta=0.019,sig=0.61,rho=-0.7,r=0.0319,
                      K=100,f1=fc,f2=fc,f3=fb)

         res_pt = Monte_Carlo8(n_=1000,m_=10000,T=1,S0=100,V0=0.010201,
```

```
                    k=6.21,theta=0.019,sig=0.61,rho=-0.7,r=0.0319,
                    K=100,f1=fc,f2=fc,f3=fa)

        res_ful = Monte_Carlo8(n_=1000,m_=10000,T=1,S0=100,V0=0.010201,
                    k=6.21,theta=0.019,sig=0.61,rho=-0.7,r=0.0319,
                    K=100,f1=fc,f2=fa,f3=fa)
```

```
In [95]: result_tablebo1 = pd.DataFrame([[res_ab[0],res_ab[1],res_ab[2],res_ab[3]],
                                        [res_re[0],res_re[1],res_re[2],res_re[3]],
                                        [res_hm[0],res_hm[1],res_hm[2],res_hm[3]],
                                        [res_pt[0],res_pt[1],res_pt[2],res_re[3]],
                                        [res_ful[0],res_ful[1],res_ful[2],res_ful[3]]],
                                       index = ["Absorption","Reflection","Higham and Mao","Pa
                                       columns = ["price","bias","RMSE","Time consumed (s)"])
        print(result_tablebo1)
```

```
                      price      bias      RMSE  Time consumed (s)
Absorption         6.776724  0.029376  0.075418          80.841880
Reflection         7.084331  0.278231  0.081001           7.239646
Higham and Mao     6.960910  0.154810  0.076532           7.013221
Partial truncation 6.857270  0.051170  0.074215           7.239646
Full truncation    6.786970  0.019130  0.073660          77.698319
```

As we can see, the results are near to the exact price given which is 6.8061.

4.2

Here we use the quadrature method to price this option. Note that in this model $\lambda = 0$ according to the paper said.

But first of all, we need to contruct the Simpson quadratic rule.

```
In [106]: def simpson_int(func,a,b,tol):
              n=10000
              delta = (b-a)/n
              x = np.linspace(a,b,n+1)
              f_x = np.asarray([func(i) for i in x])
              res0 = 0
              res1 = delta/3*(f_x[0]+f_x[-1]+4*f_x[1:-1][::2].sum()+2\
                              *f_x[1:-1][1::2].sum())
              while abs(res1-res0)>tol:
                  n= n + 10000
                  x = np.linspace(a,b,n+1)
                  f_x = np.asarray([func(i) for i in x])
                  delta = (b-a)/n
                  res0 = res1
                  res1 = delta/3*(f_x[0]+f_x[-1]+4*f_x[1:-1][::2].sum()+2*\
                                  f_x[1:-1][1::2].sum())
              return res1
```

Then we contruct the function to use integral to get the option's price.

```
In [107]: def C_integral(tau,S0,V0,k,theta,sig,rho,r,K):
              u1 = 0.5
              u2 = -0.5
              a = k*theta
              b1 = k-rho*sig
              b2 = k

              def f1(u):
                  com = np.complex(b1,-rho*sig*u)
                  d1 = np.sqrt((-com)**2-sig**2*(np.complex(0,2*u1*u)-u**2))
                  g1 = (com+d1)/(com-d1)
                  C1 = np.complex(0,r*u*tau)+a/sig**2*((com+d1)*tau-\
                          2*np.log((1-g1*np.exp(d1*tau))/(1-g1)))
                  D1 = (com+d1)/sig**2*((1-np.exp(d1*tau))/(1-g1*np.exp(d1*tau)))
                  phi1 = np.exp(C1+D1*V0+np.complex(0,u*np.log(S0)))
                  res = ((np.exp(np.complex(0,-np.log(K)*u))*\
                          phi1)/(np.complex(0,u))).real
                  return res
              def f2(u):
                  com = np.complex(b2,-rho*sig*u)
                  d2 = np.sqrt((-com)**2-sig**2*(np.complex(0,2*u2*u)-u**2))
                  g2 = (com+d2)/(com-d2)
                  C2 = np.complex(0,r*u*tau)+a/sig**2*((com+d2)*tau\
                          -2*np.log((1-g2*np.exp(d2*tau))/(1-g2)))
                  D2 = (com+d2)/sig**2*((1-np.exp(d2*tau))/(1-g2*np.exp(d2*tau)))
                  phi2 = np.exp(C2+D2*V0+np.complex(0,u*np.log(S0)))
                  res = ((np.exp(np.complex(0,-np.log(K)*u))*\
                          phi2)/(np.complex(0,u))).real
                  return res
              P1 = 0.5+simpson_int(f1,0.0001,1000,1e-4)/np.pi
              P2 = 0.5+simpson_int(f2,0.0001,1000,1e-4)/np.pi
              result = S0*P1-K*np.exp(-r*tau)*P2
              return result
```

```
In [108]: res_int = C_integral(tau=1,S0=100,V0=0.010201,
                       k=6.21,theta=0.019,sig=0.61,rho=-0.7,r=0.0319,
                       K=100)
```

```
In [109]: print(res_int)
```

```
8.479037160469325
```

The price we compute here is almost $8.5.

4.3

Comparing two methods we find the first method although involve the random variable get a good estimation for the price. The second method is less accurate the potential reson may be the

upper limit of quadratic is small. However the thing is that when we increase this upper limit, the python may encounter overflow of numpy. This issue still exist when we scal the parameter into a small one.

For the small parameter given in the paper, we can give a relative accurate value.