Computational Finance_Project 2

January 24, 2019

In [ ]: Xiangui Mei

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math as math
from scipy.stats import norm
import random as random
```

In [ ]: Q1 Find corrcoef of bivariate-normally distributed random vectors

In [2]:
```
def bi_var(seed):
    n=1000
    np.random.seed([seed])
    # first simulate 2 independent random normals
    # then get x and y
    Z1=np.random.normal(0,1,n)
    Z2=np.random.normal(0,1,n)
    corr=-0.7
    mu1=mu2=0
    x=mu1+Z1
    y=mu2+corr*Z1+np.sqrt(1-corr*corr)*Z2
    #to get the rho
    sd_x=np.std(x,ddof=1)
    sd_y=np.std(y,ddof=1)
    nomi=(1.0/(n-1.0))*np.dot((x-np.mean(x)),(y-np.mean(y)))
    rho=nomi/(sd_x*sd_y)
    return (rho)
bi_var(123)
```

Out[2]: -0.6974708914011742

In [ ]: Q2 Evaluate the expected values by using Monte Carlo simulation

In [4]:
```
def mc_val(seed):
    n=10000
    np.random.seed([seed])
    # first simulate 2 independent random normal
    corr=0.6
```

```
        mu1=mu2=0
        z1=np.random.normal(0,1,n)
        z2=np.random.normal(0,1,n)
        X=mu1+z1
        Y=mu2+corr*z1+np.sqrt(1-corr*corr)*z2
        #use Monte Carlo Simulation
        E=np.zeros(n)
        for i in range(n):
            E[i]=max(0,(pow(X[i],3)+math.sin(Y[i])+(pow(X[i],2))*Y[i]))
        return (np.mean(E))
    mc_val(121)
```

Out[4]: 1.4863115312862227

In [ ]: Q3 (a) Estimate the following expected values by simulation

In [7]:

```
# standard Wiener Process follows normal distribution of N(0,sqrt(t))
n=10000
z1=np.random.normal(0,1,n)
# define Wiener Process
# W5 simulation
def w(t):
    w=np.sqrt(t)*z1
    return w
w5=w(5)
F1=np.zeros(n)
for i in range(n):
    F1[i]=w5[i]*w5[i]+math.sin(w5[i])
print("mean(Ea1) and variance are separatly:")
print (round(np.mean(F1),4),round(np.var(F1),4))

# Wt simulation
def F2(t):
    w=np.zeros(n)
    F2=np.zeros(n)
    for i in range(n):
        w[i]=np.sqrt(t)*z1[i]
        F2[i]=np.exp(t/2)*math.cos(w[i])
    return(F2)
print ("mean for t=0.5(Ea2),3.2(Ea3) and 6.5(Ea4) are separately:")
print(round(np.mean(F2(0.5)),4),round(np.mean(F2(3.2)),4),+
      round(np.mean(F2(6.5)),4))
print ("variance for t=0.5,3.2 and 6.5 are separately:")
print(round(np.var(F2(0.5)),4),round(np.var(F2(3.2)),4),+
      round(np.var(F2(6.5)),4))
```

```
mean(Ea1) and variance are separatly:
(4.9927, 49.8155)
mean for t=0.5(Ea2),3.2(Ea3) and 6.5(Ea4) are separately:
```

```
(1.0005, 0.9945, 0.9417)
variance for t=0.5,3.2 and 6.5 are separately:
(0.1265, 11.2467, 327.8284)
```

In [ ]:
```
Q3 (b) How are the values of the last three integrals related
```

In [ ]:
```
The pratical value for last three integrals are very close to 1.
The theoretical value for last three integrals are same, which is 1.
Proof: If Wt is a standard Wiener Process, we know that Wt
is normally distributed with mean 0 and variance t.
e^(t/2)cos(Wt)=1+integral(-e^(t/2)/2*sin(Wt)dWt: from 0 to 1)
Taking the mean,
E(e^(t/2)cos(Wt))=E(1+integral(-e^(t/2)/2*sin(Wt)dWt: from 0 to 1))
Since integral(sin(Wt)dWt) is a martingale,
E(e^(t/2)cos(Wt))=1
So no matter what t is, the integral are close to 1.
```

In [ ]:
```
Q3 (c) Now use a variance reduction technique (whichever you want)
to compute the expected values in part (a)
```

In [9]:
```
# use control variate method to reduce variance
Y1=Y2=Y3=Y4=np.zeros(n)
T1=np.zeros(n)
for i in range(n):
    Y1[i]=w5[i]*w5[i]
gamma1=np.cov(Y1,F1)[1,0]/np.var(Y1)
T1=F1-gamma1*(Y1-np.mean(Y1))
print("mean(Eb1) and variance after variance reduction are separatly:")
print (round(np.mean(T1),4),round(np.var(T1),8))

T2=T3=T4=np.zeros(n)
for i in range(n):
    Y2[i]=w(0.5)[i]*w(0.5)[i]
    Y3[i]=w(3.2)[i]*w(3.2)[i]
    Y4[i]=w(6.5)[i]*w(6.5)[i]
gamma2=np.cov(Y2,F2(0.5))[1,0]/np.var(Y2)
gamma3=np.cov(Y3,F2(3.2))[1,0]/np.var(Y3)
gamma4=np.cov(Y4,(F2(6.5)))[1,0]/np.var(Y4)
T2=F2(0.5)-gamma2*(Y2-np.mean(Y2))
T3=F2(3.2)-gamma3*(Y3-np.mean(Y3))
T4=F2(6.5)-gamma4*(Y4-np.mean(Y4))
print("means for t=0.5(Eb2),3.2(Eb3) and 6.5(Eb4) are separately:")
print(round(np.mean(T2),4),round(np.mean(T3),4),round(np.mean(T4),4))
print ("variance for t=0.5,3.2 and 6.5 are separately:")
print(round(np.var(T2),4),round(np.var(T3),4),round(np.var(T4),4))
```

```
mean(Eb1) and variance after variance reduction are separatly:
(4.9927, 0.50497651)
```

```
means for t=0.5(Eb2),3.2(Eb3) and 6.5(Eb4) are separately:
(1.0005, 0.9945, 0.9417)
variance for t=0.5,3.2 and 6.5 are separately:
(0.0025, 6.0749, 306.4137)
```

In [ ]: After using the control variate method, the estimated value is closer to the true value, the variances are also smaller.

In [ ]: Q4 (a) Estimate the price c of a European Call option by MC Simulation

In [28]:
```python
T = 5.0
S0 = 88.0
r = 0.04
sd = 0.2
K = 100.0
S_p=S0*np.exp(sd*w(T)+(r-(sd*sd)/2)*T)
C=np.zeros(10000)
for i in range(10000):
    if S_p[i]>K:
        C[i]=S_p[i]-K
    else:
        C[i]=0
print("The price of call option Ca1 is:")
print(round(np.exp(-r*T)*np.mean(C),4))
print("The variance of Ca1 is:")
print(round(np.var(np.exp(-r*T)*C),4))
```

```
The price of call option Ca1 is:
18.5391
The variance of Ca1 is:
1074.2717
```

In [ ]: Q4(b) Compute the exact value of the option c by the BS formula.

In [11]:
```python
# with Black-Sholes Formula
def euro_bs_call(S,K,T,r,sd):
    d1 = (np.log(S/K) + (r + 0.5 * sd ** 2) * T) / (sd * np.sqrt(T))
    d2 = d1 - sd*np.sqrt(T)
    Cb1 = S0*norm.cdf(d1) - K*np.exp(-r*T)*norm.cdf(d2)
    return(Cb1)
print("The call option price with Black-Sholes Formula Cb1 is:")
print(round(euro_bs_call(88.0,100.0,5.0,0.04,0.2),4))
```

```
The call option price with Black-Sholes Formula Cb1 is:
18.2838
```

```
In [ ]: Q4 (b) Now use variance reduction techniques to estimate the
        price in part (a) again
```

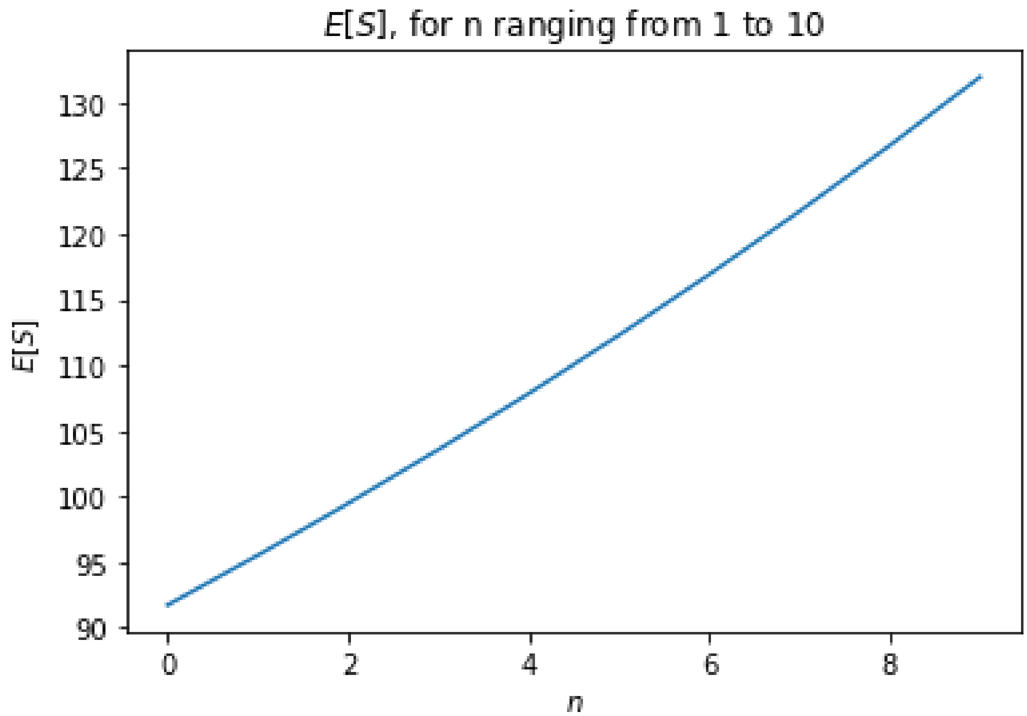
```
In [12]: # with variance reduction method
         # using antithetic variates method
         S_n=S0*np.exp(sd*w(T)*(-1)+(r-(sd*sd)/2)*T)
         C_n=np.zeros(10000)
         for i in range(10000):
             if S_n[i]>K:
                 C_n[i]=S_n[i]-K
             else:
                 C_n[i]=0
         Cb2=np.exp(-r*T)*(0.5*np.mean(C_n)+0.5*np.mean(C))
         print("The price of call option Cb2 is:")
         print(round(Cb2,4))
         print("The variance of Cb2 is:")
         print(round(np.var(np.exp(-r*T)*(0.5*C_n+0.5*C)),4))
```

```
The price of call option Cb2 is:
18.2743
The variance of Cb2 is:
342.6221
```

```
In [ ]: Comment: The call price after variance reduction is closer to the theoritical value
        , and the variance reduced a lot from 1074.2717 to 342.6221.
```

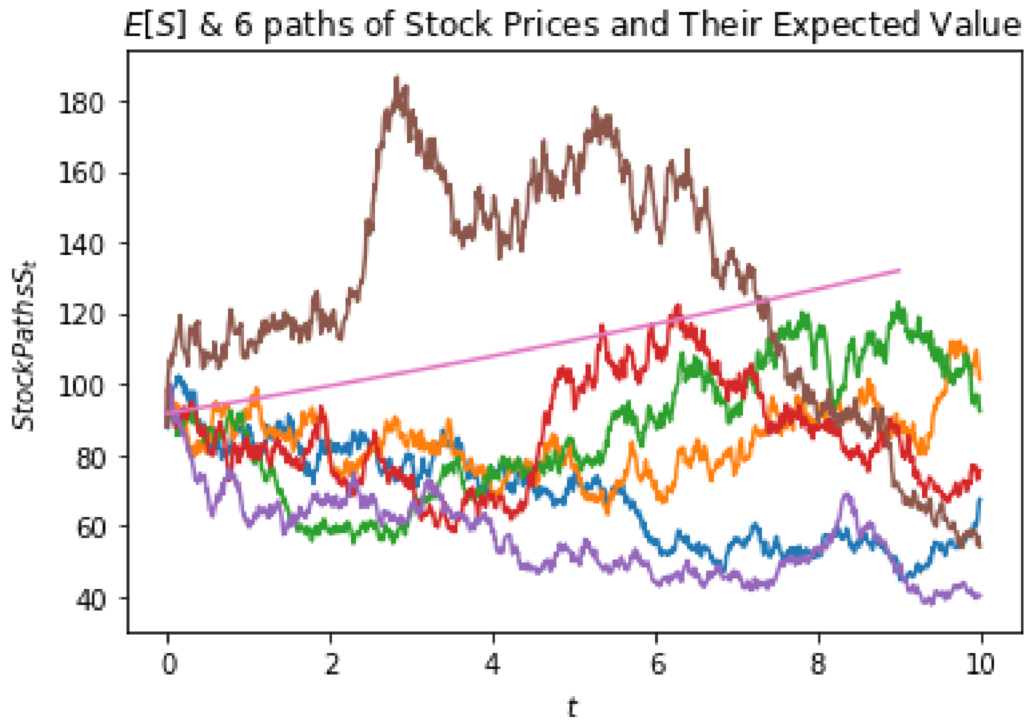
```
In [ ]: Q5 (a) Plot all of the above E(Sn)
```

```
In [13]: # simulate the stock process
         n_5 = 1000
         S0_5 = 88.0
         sd_5 = 0.18
         r_5 = 0.04
         ES=np.zeros(10)
         for i in range (1,11):
             S_t=np.zeros(n)
             S_t= S0_5*np.exp(sd_5*w(i)+(r_5-(sd_5*sd_5)/2)*i)
             ES[i-1]=np.mean(S_t)

         plt.plot(ES)
         plt.title('$E[S]$, for n ranging from 1 to 10')
         plt.xlabel('$n$')
         plt.ylabel('$E[S]$')
         plt.show()
```




```
In [ ]: Q5 (b) Now simulate 6 paths of St for t in [0:10]
```

```
In [15]: n5=1000
         dt = np.sqrt(10.0/n5)
         path = np.zeros((6,n5+1))
         wdt = np.zeros((10,n5))

         plt.figure
         for i in range(6):
             Zi = np.random.normal(0,1,n5)
             wdt[i,0] = dt*Zi[0]
             for j in range(1,n5):
                 wdt[i,j] = wdt[i,j-1] + dt*np.array(Zi[j])
             path[i,1:(n+1)] = S0*np.exp(sd_5*wdt[i,:]+(r-(sd_5**2/2))*(i+1))
             path[i,0] = S0
             plt.plot(np.arange(0,10,10.0/1001),path[i,:])
         plt.plot(ES)
         plt.title("$E[S]$ & 6 paths of Stock Prices and Their Expected Value")
         plt.xlabel("$t$")
         plt.ylabel("$Stock Paths S_t$")
         plt.show()
```



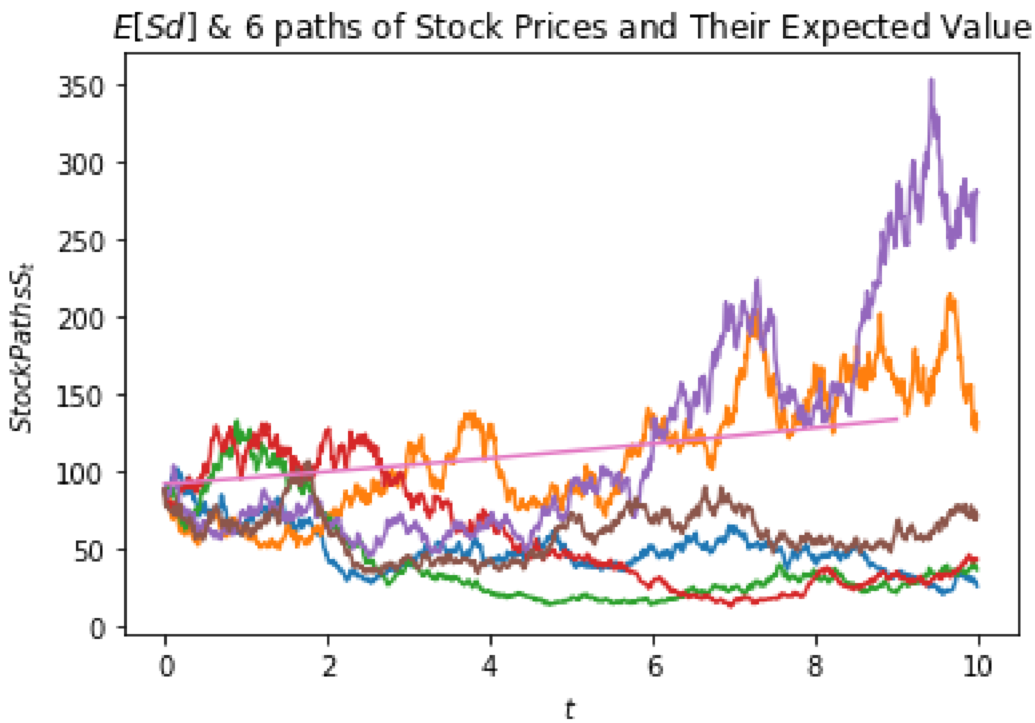
In [ ]: Q5 (c) Plot your data from parts (a) and (b) in one graph.

In [ ]: The graph is plotted in Q5(b)

In [ ]: Q5 (d) Now the std= 35%

In [24]:
```
# under sd=35%
sd_52=0.35
ESd=np.zeros(10)
for i in range (1,11):
    S_t=np.zeros(n5)
    S_t= S0_5*np.exp(sd_52*w(i)+(r_5-(sd_52*sd_52)/2)*i)
    ESd[i-1]=np.mean(S_t)

path2 = np.zeros((6,n5+1))
plt.figure
for i in range(6):
    Zi = np.random.normal(0,1,n5)
    wdt[i,0] = dt*Zi[0]
    for j in range(1,n5):
        wdt[i,j] = wdt[i,j-1] + dt*np.array(Zi[j])
    path2[i,1:(n5+1)] = S0*np.exp(sd_52*wdt[i,:]+(r-(sd_52**2/2))*(i+1))
    path2[i,0] = S0
    plt.plot(np.arange(0,10,10.0/1001),path2[i,:])
```

```
plt.plot(ESd)
plt.title("$E[Sd]$ & 6 paths of Stock Prices and Their Expected Value")
plt.xlabel("$t$")
plt.ylabel("$Stock Paths S_t$")
plt.show()

print("The graph for E[S] and E[Sd]  didn't change at all. ")
print("But with a higher std,6 paths are much more volatile")
```




```
The graph for E[S] and E[Sd]  didn't change at all.
But with a higher std,6 paths are much more volatile
```

```
In [ ]: Q6 (a) Euler's discretization
```

```
In [18]: n6=1001
         x6=np.linspace(0,1,n6)
         delta=1.0/(n6-1)
         y6=0
         for i in range(1,n6):
            y6+=np.sqrt(1-x6[i]*x6[i])*delta*4
         print("By using Euler's Method,the integration is:" )
         print(round(y6,4))
```

```
By using Euler's Method,the integration is:
3.1396
```

In [ ]: Q6 (b) Monte Carlo Simulation

In [19]:
```python
x6b=np.random.uniform(0,1,n6)
y6b=np.zeros(n6)
for i in range(1,n6):
    y6b[i]=np.sqrt(1-x6b[i]*x6b[i])*4
print("By using Monte Carlo Simulation,the integration is:" )
print(round(np.mean(y6b),4))
```

```
By using Monte Carlo Simulation,the integration is:
3.1577
```

In [ ]: Q6 (c) Importance Sampling method

In [20]:
```python
n6c=10000
random.seed(123)
y=[]
u=np.random.uniform(0,1,n6c)

random.seed(121)
u2=np.random.uniform(0,1,n6c)

h = (1-0.74*(np.array(u)**2))/(1-0.74/3.0)

for i in range(1000):
    if u2[i] <= h[i]/1.5:
       y.append(u[i])

h_y= (1-0.74*(np.array(y)**2))/(1-0.74/3.0)
g_y= np.sqrt(1-np.array(y)**2)
sim_i=4*g_y*1/h_y
print("By using Importance Sampling method ,the integration is:" )
print(round(np.mean(sim_i),4))
```

```
By using Importance Sampling method ,the integration is:
3.1355
```

In [ ]:
```
Comment: Compared with Monte Carlo Simulation, the accuracy
for Importance Sampling method improved as the estimated value is
closer to the true value. But overall, Euler's discretization has the best
performance.
```