Computational Finance_Project 3

January 31, 2019

```
In [ ]: Xiangui Mei
```

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

```
In [ ]: Q1 Evaluate the following expected values and probabilities
```

```
In [2]: # Find Yt and Xt:
        # standard Wiener Process follows normal distribution of N(0,sqrt(t))
        def Xt(time,steps,sed):
            Xt_current=1.0
            dt=1.0/steps
            X_t=0.0
            t=0.0
            for i in range(time*steps):
                random.seed(sed)
                dWt=m.sqrt(dt)* np.random.normal(0,1)
                dXt=(1.0/5.0)*dt-(1.0/2.0)*Xt_current*dt+(2.0/3.0)*dWt
                X_t=Xt_current+dXt
                t=t+dt
                i=i+1
                Xt_current=X_t
            return X_t


        def Yt(time,steps,sed):
            Yt_current=3.0/4.0
            dt=1.0/steps
            Y_t=0.0
            t=0.0
            for i in range(time*steps):
                random.seed(sed)
                dZt=m.sqrt(dt)* np.random.normal(0,1)
                dYt=(2.0/(1+t))*Yt_current*dt+((t**3+1.0)/3.0)*dt+((t**3 +1.0)/3.0)*dZt
                Y_t=Yt_current + dYt
```

```
            t=t+dt
            i=i+1
            Yt_current=Y_t
        return Y_t
```

In [ ]: P(Y2>5)

```
In [3]: # P(Y2>5)
        Y_2=np.zeros(100)
        a=0
        for j in range(100):
            Y_2[j]=Yt(2,100,2)
            if Y_2[j]>5:
                a=a+1
        print("The prob that Y2 is larger than 5 is:" )
        print(a/100.0)
```

```
The prob that Y2 is larger than 5 is:
0.96
```

In [ ]: E(X2^(1/3))

```
In [8]: # E(X2^(1/3))
        fX_2=np.zeros(100)
        for j in range(100):
            fX_2[j]=np.sign(Xt(2,100,123))*np.absolute(Xt(2,100,0.1))**(1.0/3.0)
        print("The expected value of fX_2 (E1) is:")
        print(round(np.mean(fX_2),4))
```

```
The expected value of fX_2 (E1) is:
0.5442
```

In [ ]: E(Y3)

```
In [7]: # E(Y3)
        Y_3=np.zeros(100)
        for j in range(100):
            Y_3[j]=Yt(3,100,121)
        print("The expected value of Y3 (E2) is:")
        print(round(np.mean(Y_3),4))
```

```
The expected value of Y3 (E2) is:
25.7274
```

In [ ]: E(X2Y21(X2>1))

In [9]:
```
# E(X2Y2(X2>1))
XY_2=np.zeros(200)
for j in range(200):
    X2=Xt(2,100,123)
    Y2=Yt(2,100,321)
    if X2>1.0:
        XY_2[j]=X2*Y2
    else:
        XY_2[j]=0.0
print("The expected value of X2Y2 (E3) is:")
print(round(np.mean(XY_2),4))
```

```
The expected value of X2Y2 (E3) is:
4.0663
```

In [ ]: Q2 Estimate the following expected values:

In [10]:
```
# Find xt and yt
def xt(time,steps,sed1,sed2):
    xt_current=1.0
    dt=1.0/steps
    x_t=0.0
    t=0.0
    for i in range(time*steps):
        #get two independent wiener process
        random.seed(sed1)
        dwt=m.sqrt(dt)* np.random.normal(0,1)
        random.seed(sed2)
        dzt=m.sqrt(dt)* np.random.normal(0,1)
        #get dxt
        dxt=(1.0/4.0)*xt_current*dt+(1.0/3.0)*xt_current*dwt \
        -(3.0/4.0)*xt_current*dzt
        x_t=xt_current+dxt
        t=t+dt
        i=i+1
        xt_current=x_t
    return x_t

def yt(t,sed1,sed2):
        #get two independent wiener process
        random.seed(sed1)
        wt=m.sqrt(t)* np.random.normal(0,1)
        random.seed(sed2)
        zt=m.sqrt(t)* np.random.normal(0,1)
        #get yt
        y_t=np.exp(-0.08*t+(1.0/3.0)*wt+(3.0/4.0)*zt)
        return y_t
```

In [ ]: E((1+X3)^(1/3))

```
In [13]: # E((1+X3)^(1/3))
         fx_3=np.zeros(100)
         for j in range(100):
             xt_1=1.0+xt(2,100,123,121)
             fx_3[j]=np.sign(xt_1)*np.absolute(xt_1)**(1.0/3.0)
         print("The expected value of fx_3 (E1) is:")
         print(round(np.mean(fx_3),4))
```

```
The expected value of fx_3 (E1) is:
1.3096
```

In [ ]: E((1+Y3)^(1/3))

```
In [14]: # E((1+Y3)^(1/3))
         fy_3=np.zeros(100)
         for j in range(100):
             yt_1=1.0+yt(3,121,123)
             fy_3[j]=np.sign(yt_1)*np.absolute(yt_1)**(1.0/3.0)
         print("The expected value of fy_3 (E2) is:")
         print(round(np.mean(fy_3),4))
```

```
The expected value of fy_3 (E2) is:
1.3569
```

In [ ]: Q3 (a) Write code to compute the prices of European Call options
via Monte Carlo simulation.

```
In [46]: # European Call options via Monte Carlo Simulation
         # With antithetic variates, we need to generate wt and -wt
         def eur_anti_call(S_0,X,r,sigma,T):
             WT=m.sqrt(T)* np.random.normal(0,1,10000)
             # generate a positive C_p
             S_p=S_0*np.exp(sigma*np.array(WT)+(r-(sigma*sigma)/2)*T)
             C_p=np.zeros(10000)
             for i in range(10000):
                 if S_p[i]>X:
                    C_p[i]=S_p[i]-X
                 else:
                    C_p[i]=0
             #generate a negative C_n
             S_n=S_0*np.exp(sigma*np.array(WT)*(-1)+(r-(sigma*sigma)/2)*T)
             C_n=np.zeros(10000)
             for i in range(10000):
                 if S_n[i]>X:
                    C_n[i]=S_n[i]-X
```

```
            else:
                C_n[i]=0
        # taking the average of C_p and C_n
        C1=np.exp(-r*T)*(0.5*np.mean(C_n)+0.5*np.mean(C_p))
        return C1
```

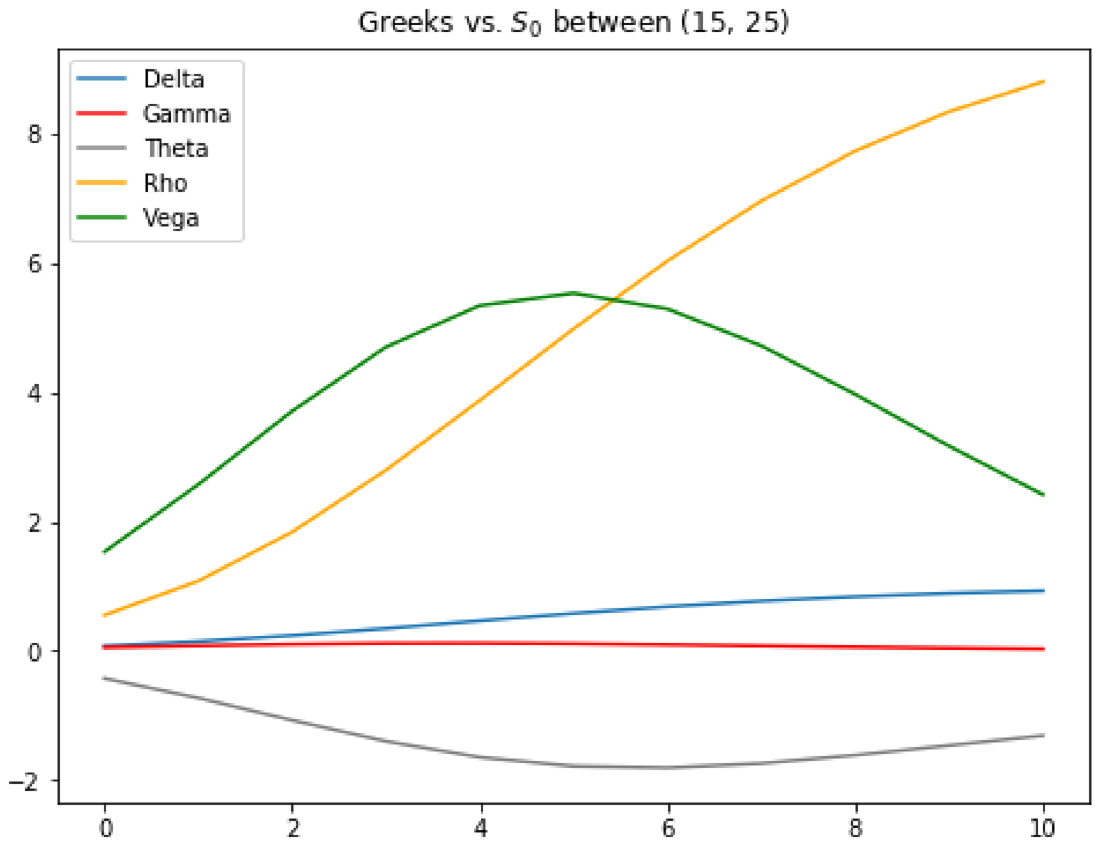
In [ ]: Q3 (b) Write code to compute the prices of European Call options by using the Black-Scholes

In [45]:
```
# the approximation of N(x)
def N(x):
    d1 = 0.0498673470
    d2 = 0.0211410061
    d3 = 0.0032776263
    d4 = 0.0000380036
    d5 = 0.0000488906
    d6 = 0.0000053830
    if x >= 0:
        N_x=1-(1.0/2.0)*(1+d1*x+d2*x**2+d3*x**3 \
                        + d4*x**4+d5*x**5+d6*x**6)**(-16)
    else:
        N_x=1-(1-(1.0/2.0)*(1+d1*(-x)+d2*(-x)**2+d3*(-x)**3 \
                        +d4*(-x)**4+d5*(-x)**5+d6*(-x)**6)**(-16))
    return N_x

# With B-S Formula
def euro_bs_call(S_0,X,r,sigma,T):
    d1 = (np.log(S_0/X)+(r+ 0.5 * sigma ** 2)*T)/(sigma * np.sqrt(T))
    d2 = d1 - sigma*np.sqrt(T)
    C2 = S_0*N(d1) - X*np.exp(-r*T)*N(d2)
    return C2
```

In [ ]: Q3 (c) Estimate the hedging parameters of European Call options (all five Greeks) and graph them.

In [25]:
```
# define fuction for greeks
X=20.0
sigma=0.25
r=0.04
T=0.5
h=0.1
S_0 = [i for i in range(15,26)]
S0_ph=np.array(S_0)+h
S0_nh=np.array(S_0)-h
# delta
delt=np.zeros(11)
for j in range (0,11):
    delt[j]=(euro_bs_call(S0_ph[j],X,r,sigma,T)- \
             euro_bs_call(S0_nh[j],X,r,sigma,T))/(2*h)
```

```
# gamma
gamma=np.zeros(11)
nomi1=np.zeros(11)
nomi2=np.zeros(11)
for j in range (0,11):
    nomi1[j]= euro_bs_call(S0_ph[j],X,r,sigma,T)+ \
    euro_bs_call(S0_nh[j],X,r,sigma,T)
    nomi2[j]= 2*euro_bs_call(S_0[j],X,r,sigma,T)
    gamma[j]=(nomi1[j]-nomi2[j])/(h*h)
    #gamma is the soc derivative
# theta
dt=0.001
theta=np.zeros(11)
for j in range (0,11):
    theta[j]=(euro_bs_call(S_0[j],X,r,sigma,T+dt)- \
              euro_bs_call(S_0[j],X,r,sigma,T-dt))/(-2*dt)
# vega
dv=0.001
vega=np.zeros(11)
for j in range (0,11):
    vega[j]=(euro_bs_call(S_0[j],X,r,sigma+dv,T)- \
             euro_bs_call(S_0[j],X,r,sigma-dv,T))/(2*dv)
# rho
rho=np.zeros(11)
for j in range (0,11):
    rho[j]=(euro_bs_call(S_0[j],X,r+h,sigma,T)- \
            euro_bs_call(S_0[j],X,r-h,sigma,T))/(2*h)

plt.figure(figsize=(8,6))
plt.plot(delt,label="Delta")
plt.plot(gamma,color='red',label="Gamma")
plt.plot(theta,color="grey",label="Theta")
plt.plot(rho,color="orange",label="Rho")
plt.plot(vega,color="green",label="Vega")
plt.title("Greeks vs. $S_0$ between (15, 25)")
plt.legend()
plt.show()
```



In [ ]: Q4 Use the Full Truncation, Partial Truncation and Reflection methods, and provide 3 price estimates by using the tree methods.

In [43]:
```
# find a bi-variate random variable
def bi_var(n):
    random.seed(2)
    z1=np.random.normal(0,1,n)
    z2=np.random.normal(0,1,n)
    corr=-0.6
    mu1=mu2=0
    Z1=mu1+z1
    Z2=mu2+corr*z1+np.sqrt(1-corr*corr)*z2
    return [Z1,Z2]
# simulate the price of Stock (Partical Reflection)
def St(T,steps,alpha,rf,beta,sigma,Scheme):
    St_current=48.0
    Vt_current=0.05
    dt=T/steps
    dW1=m.sqrt(dt)* bi_var(steps)[0]
    dW2=m.sqrt(dt)* bi_var(steps)[1]
```

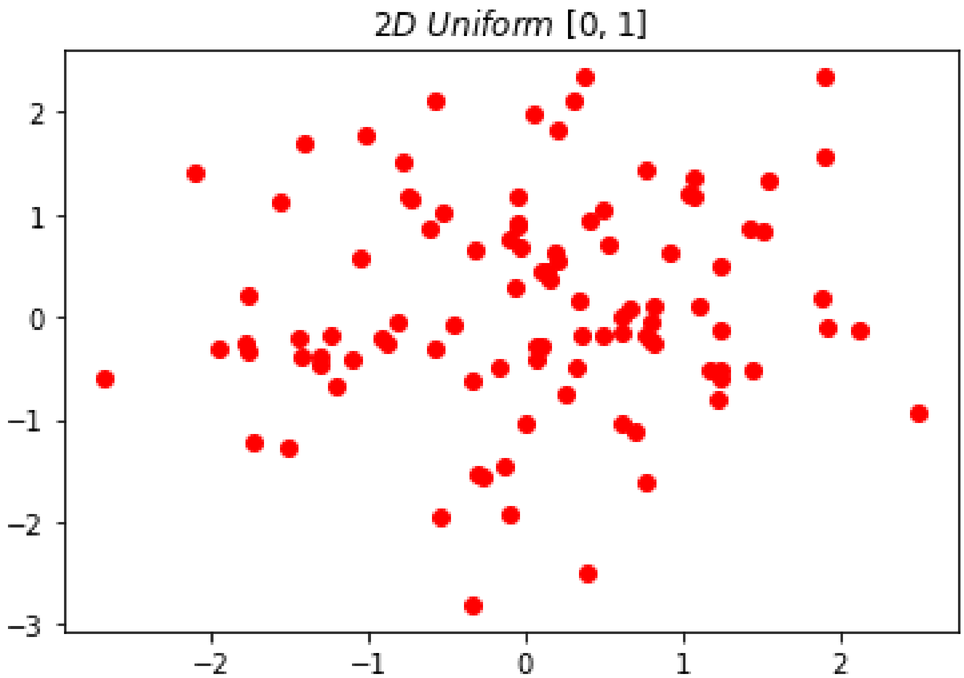
```
    if Scheme=="Full_trunction":
        for i in range(int(steps)):
            dSt=rf*St_current*dt+np.sqrt(max(Vt_current,0))*St_current*dW1[i]
            St=St_current+dSt
            dVt=alpha*(beta-max(Vt_current,0))*dt+sigma* \
            np.sqrt(max(Vt_current,0))*dW2[i]
            Vt=Vt_current+dVt
            St_current=St
            Vt_current=Vt
    elif Scheme=="Reflection":
        for i in range(int(steps)):
            dSt=rf*St_current*dt+np.sqrt(np.absolute(Vt_current))*St_current*dW1[i]
            St=St_current+dSt
            dVt=alpha*(beta-np.absolute(Vt_current))*dt+sigma* \
            np.sqrt(np.absolute(Vt_current))*dW2[i]
            Vt=Vt_current+dVt
            St_current=St
            Vt_current=Vt
    elif Scheme=="Partical_trunction":
        for i in range(int(steps)):
            dSt=rf*St_current*dt+np.sqrt(max(Vt_current,0))*St_current*dW1[i]
            St=St_current+dSt
            dVt=alpha*(beta-Vt_current)*dt+sigma* \
            np.sqrt(max(Vt_current,0))*dW2[i]
            Vt=Vt_current+dVt
            St_current=St
            Vt_current=Vt
    else:
        for i in range(int(steps)):
            dSt=rf*St_current*dt+np.sqrt(Vt_current)*St_current*dW1[i]
            St=St_current+dSt
            dVt=alpha*(beta-Vt_current)*dt+sigma*np.sqrt(Vt_current)*dW2[i]
            Vt=Vt_current+dVt
            St_current=St
            Vt_current=Vt
    return St

# By using Monte Carlo Simulation with Full_trunction
S_ft=np.zeros(1000)
C_ft=np.zeros(1000)
X=50.0
for i in range(1000):
    S_ft[i]= St(0.5,1000,5.8,0.03,0.0625,0.42,"Full_trunction")
    if S_ft[i]>X:
       C_ft[i]=S_ft[i]-X
    else:
       C_ft[i]=0
print("The price of Full_trunction call option C1 is:")
```

```
print(round(np.exp(-0.03*0.5)*np.mean(C_ft),4))

# By using Monte Carlo Simulation with Partical_trunction
S_pt=np.zeros(1000)
C_pt=np.zeros(1000)
X=50.0
for i in range(1000):
    S_pt[i]= St(0.5,1000,5.8,0.03,0.0625,0.42,"Partical_trunction")
    if S_pt[i]>X:
       C_pt[i]=S_pt[i]-X
    else:
       C_pt[i]=0
print("The price of Partical_trunction call option C2 is:")
print(round(np.exp(-0.03*0.5)*np.mean(C_pt),4))

# By using Monte Carlo Simulation with Reflection
S_r=np.zeros(1000)
C_r=np.zeros(1000)
X=50.0
for i in range(1000):
    S_r[i]= St(0.5,1000,5.8,0.03,0.0625,0.42,"Reflection")
    if S_r[i]>X:
       C_r[i]=S_r[i]-X
    else:
       C_r[i]=0
print("The price of Reflection call option C3 is:")
print(round(np.exp(-0.03*0.5)*np.mean(C_r),4))
```

```
The price of Full_trunction call option C1 is:
2.6881
The price of Partical_trunction call option C2 is:
2.4303
The price of Reflection call option C3 is:
2.6561
```

In [ ]: Q5 (a) Generate 100 2-dimensional vectors of Uniform [0,1]x[0,1]

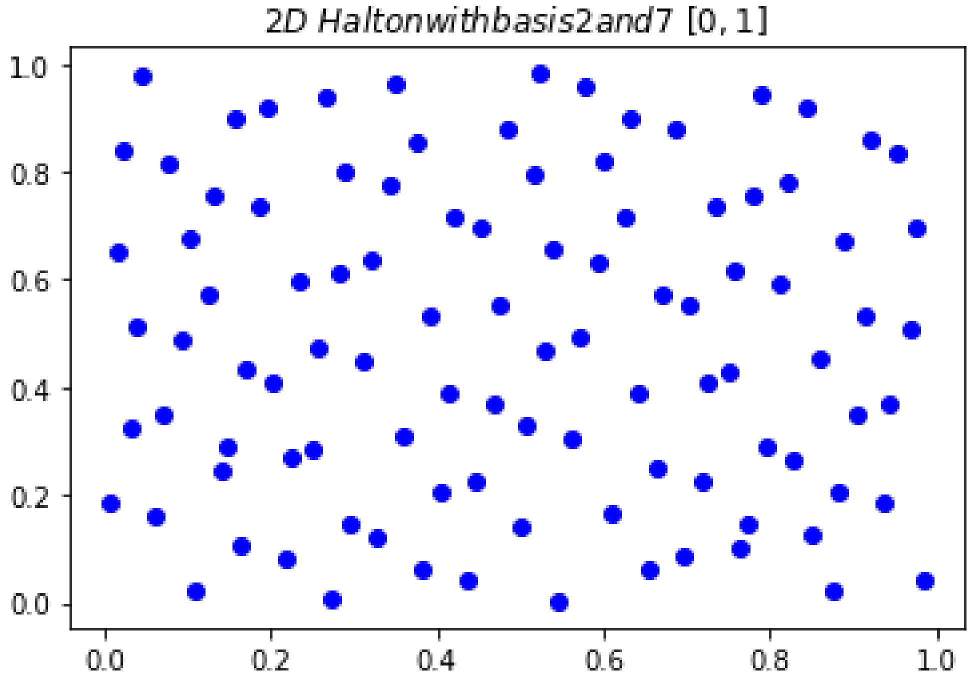
In [38]:
```
uni_2d= np.zeros((100,2))
uni_2d[:,0]=np.random.normal(0,1,100)
uni_2d[:,1]=np.random.normal(0,1,100)
plt.plot(uni_2d[:,0],uni_2d[:,1],'ro')
plt.title("$2D\ Uniform\ [0, 1]$")
plt.show()
```



In [ ]: Q5 (b) Generate 100 points of the 2-dimensional Halton sequences, basis 2 and 7

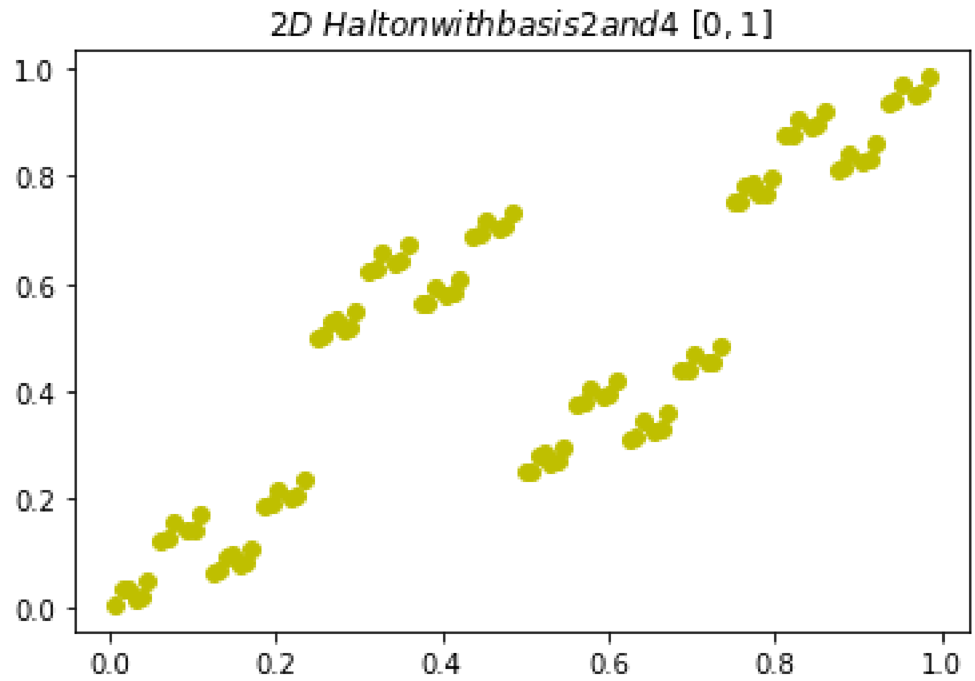
In [29]:
```
def get_Halton(base,n):
    seq=np.zeros(n)
    numbits=1+m.ceil(np.log(n)/np.log(base))
    d=np.zeros(int(numbits))
    a=np.array([i+1 for i in range(int(numbits))])
    b=1.0/base**(a)
    for i in range(n):
        j=0 ; ok =0
        while ok==0:
            d[j]=d[j]+1
            if d[j]<base:
               ok=1
            else:
               d[j]=0 ; j = j+1
        seq[i]=np.dot(d,b)
    return seq

h_2d= np.zeros((100,2))
h_2d[:,0]=get_Halton(2,100)
h_2d[:,1]=get_Halton(7,100)
plt.plot(h_2d[:,0],h_2d[:,1],'bo')
```

```
plt.title("$2D\ Halton with basis 2 and 7\ [0, 1]$")
plt.show()
```



In [ ]: Q5 (c) Generate 100 points of the 2-dimensional Halton sequences, using bases 2 and 4

In [31]:
```
h_2d1= np.zeros((100,2))
h_2d1[:,0]=get_Halton(2,100)
h_2d1[:,1]=get_Halton(4,100)
plt.plot(h_2d1[:,0],h_2d1[:,1],'yo')
plt.title("$2D\ Halton with basis 2 and 4\ [0, 1]$")
plt.show()
```



In [ ]: Q5 (d) see if there are differences in the three.

In [32]:
```
print("The graph is plotted above")
```

```
The graph is plotted above
```

In [ ]: The 2-D Uniform [0,1] sequence resembled the i.i.d uniform random numbers very well, but it is hard to predict the number since the generated numbers are unpredictable.
The 2-D Halton sequence with base 2 and 7 also demonstrates the uniform property of the i.i.d uniform distribution, when n is large, it will be very close to iid. uniform dist.
But the 2-D Halton sequence with base 2 and 4 shows strong correlation between x and y dimensions, thus it could not represent iid. uniform dist.

In [ ]: Q5 (e) Use 2-dimensional Halton sequences to compute the following integral

In [44]:
```
n = 10000
q=np.array(get_Halton(2,n))
w=np.array(get_Halton(4,n))
e=np.array(get_Halton(5,n))
r=np.array(get_Halton(7,n))

def integral(x,y):
```

```python
        f = np.zeros(n)
        for i in range(n) :
            f[i] = np.exp(-x[i]*y[i])* (m.sin(6*m.pi*x[i]) \
             + np.sign(m.cos(2*m.pi*y[i])) \
             *np.absolute(m.cos(2*m.pi*y[i]))**(1.0/3.0))
        return np.mean(f)

    print ("the result from basis 2 and 4 is")
    print(round(integral(q,w),4))
    print ("the result from basis 2 and 7 is")
    print(round(integral(q,r),4))
    print ("the result from basis 5 and 7 is")
    print(round(integral(e,r),4))
```

```
the result from basis 2 and 4 is
-0.0049
the result from basis 2 and 7 is
0.0261
the result from basis 5 and 7 is
0.0262
```