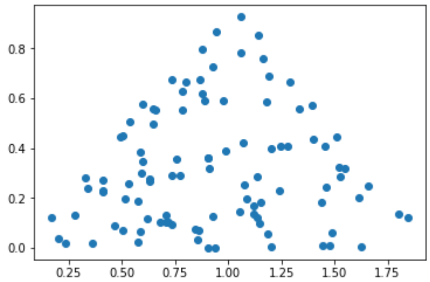
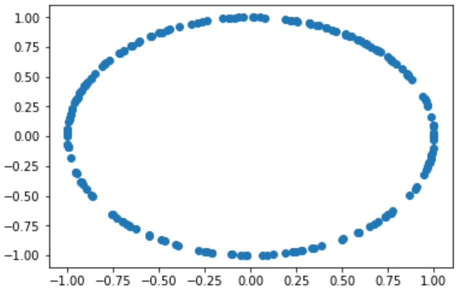
1) Inverse Transform Method to Generate Random Variables

```
In [121]: n=100
     ...: U1=np.random.rand(n)
     ...: U2=np.random.rand(n)
     ...: X=np.zeros(n)
     ...: Y=np.zeros(n)
     ...: for i in range(n):
     ...:     if U1[i]<0.5:
     ...:         X[i]=m.sqrt(2*U1[i])
     ...:         Y[i]=X[i]*U2[i]
     ...:     if U1[i]>0.5:
     ...:         X[i]=2-m.sqrt(2-2*U1[i])
     ...:         Y[i]=(2-X[i])*U2[i]
     ...:
     ...:
     ...:
     ...: plt.plot(X,Y,'o')
Out[121]: [<matplotlib.lines.Line2D at 0x296190fb4a8>]
```



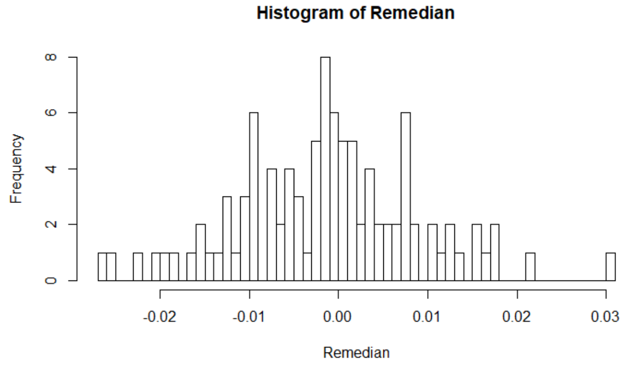
2) d-dimension unit random variables

```
In [117]: def generate(n,d):
     ...:     BX=np.zeros((n,d))
     ...:     for k in range(n):
     ...:         X=np.random.randn(d)
     ...:         U=np.zeros(d)
     ...:         length=np.sqrt(sum(X*X))
     ...:         for j in range(d):
     ...:             U[j]=X[j]/length
     ...:             BX[k,j]=U[j]
     ...:     return(BX)
     ...:
     ...:
     ...: BX=generate(200,2)
     ...: plt.plot(BX[:,0],BX[:,1],'o')
Out[117]: [<matplotlib.lines.Line2D at 0x29618fa83c8>]
```

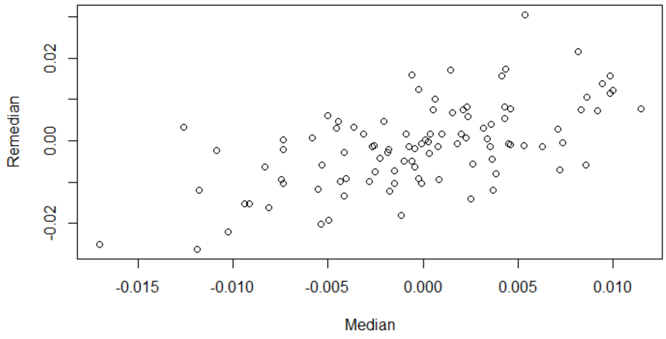


3) Remedian

```
> remedian<-function(b=15,k=4,x)
+ {
+     b1<-c()
+     b2<-c()
+     b3<-c()
+     b4<-c()
+     p=0
+     for(i in 1:b)
+     {
+         for(j in 1:b)
+         {
+             for(k in 1:b)
+             {
+                 for(l in 1:b)
+                 {
+                     p<-p+1
+                     b1[l]<-x[p]
+                 }
+                 b2[k]<-median(b1)
+             }
+             b3[j]<-median(b2)
+         }
+         b4[i]<-median(b3)
+     }
+     return(median(b4))
+ }
```




```
> plot(Remedian~Median)
```




```
> set.seed(1111)
> Remedian<-c()
> Median<-c()
> for(i in 1:100)
+ {
+     x<-rnorm(50625)
+     Remedian[i] <-remedian(15,4,x)
+     Median[i] <- median(x)
+ }
> summary(Remedian)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max. 
-0.026312 -0.007614 -0.001246 -0.001197  0.004855  0.030589 
> summary(lm(Remedian~Median))

Call:
lm(formula = Remedian ~ Median)

Residuals:
       Min         1Q     Median         3Q        Max 
-0.0158862 -0.0049369 -0.0000583  0.0046883  0.0258383 

Coefficients:
              Estimate Std. Error t value Pr(>|t|)    
(Intercept) -0.0009844  0.0008167  -1.205    0.231    
Median       1.0684235  0.1435034   7.445 3.79e-11 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.008162 on 98 degrees of freedom
Multiple R-squared:  0.3613,	Adjusted R-squared:  0.3548 
F-statistic: 55.43 on 1 and 98 DF,  p-value: 3.794e-11
```

From the above linear regression model of Remedian and True Medain, we see the F-statistic is significant. The intercept is not significant, and the coefficient is significant and approximate to 1. So we can say Remedian is nearly equal to the True Median and we can use Remedian to estimate Median directly.

4) Random Walk

```
...: n=10**6
...: First=np.zeros(n)
...: Go=np.zeros(n)
...: First[0]=1
...: obs=np.zeros((3,3))
...: k=0
...: while k<n-1:
...:
...:     U=np.random.random(1)
...:     if First[k]==1:
...:         if U<0.5:
...:             Go[k]=1
...:             obs[0,0]=obs[0,0]+1
...:         if U<5/6 and U>0.5:
...:             Go[k]=2
...:             obs[0,1]=obs[0,1]+1
...:         if U>5/6:
...:             Go[k]=3
...:             obs[0,2]=obs[0,2]+1
...:     if First[k]==2:
...:         if U<0.5:
...:             Go[k]=2
...:             obs[1,1]=obs[1,1]+1
...:         else:
...:             Go[k]=3
...:             obs[1,2]=obs[1,2]+1
...:     if First[k]==3:
...:         if U<7/8:
...:             Go[k]=1
...:             obs[2,0]=obs[2,0]+1
...:         else:
...:             Go[k]=3
...:             obs[2,2]=obs[2,2]+1
...:     First[k+1]=Go[k]
...:     k=k+1


In [59]: obs
Out[59]:
array([[222338., 149346.,  74541.],
       [     0., 148338., 149346.],
       [223886.,      0.,  32204.]])


In [61]: sum(obs)/10**6
Out[61]: array([0.446224, 0.297684, 0.256091])
```

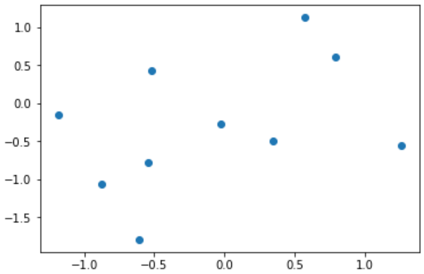
5) Newton-Raphson

```
> newtonraphson <- function(ftn, x0,tol=1e-9,max.iter=100)
+ {
+   x<-x0;
+   fx<-ftn(x)
+   iter<-0
+   while (((abs(fx[1])>tol))&&(iter<max.iter))
+   {
+     x<-x-fx[1]/fx[2]
+     fx<-ftn(x)
+     iter=iter+1
+     cat("At iteration",iter,"value of x is :",x,"\n")
+   }
+   if (abs(fx[1])>tol)
+   {
+     cat("Algorithm failed to converged\n")
+     retrun(NULL)
+
+   }
+   else
+   {
+     cat ("Algorithm converged\n ")
+     return(x)
+   }
+ }
> ftn<-function(x)
+ {
+   f=c(1/8*x**3-3/16*x**2+3/8*x-0.95,3/8*(x**2-x+1))
+   return(f)
+ }
> newtonraphson(ftn,x0=2,tol=1e-9,max.iter = 100)
At iteration 1 value of x is : 1.955556
At iteration 2 value of x is : 1.954533
At iteration 3 value of x is : 1.954532
Algorithm converged
 [1] 1.954532
```

6)

a)

```
In [94]: n=10
    ...: mu = np.array([[0, 0]]).T
    ...: Sigma = np.array([[1, 0.5], [0.5, 1]])
    ...: R = cholesky(Sigma)
    ...: s = np.dot(R,np.random.randn(2,n)) + mu
    ...: plt.scatter(s[0,:],s[1,:])
Out[94]: <matplotlib.collections.PathCollection at 0x1198b690cf8>
```




```
In [95]: s
Out[95]:
array([[-0.54432308, -0.51744262,  0.34963533,  1.26343555,  0.79325468,
         0.5720538 , -1.1850183 , -0.02806256, -0.87507593, -0.6075421 ],
       [-0.77457526,  0.42486786, -0.49585424, -0.5587643 ,  0.61373061,
         1.12671067, -0.14531576, -0.27307146, -1.05408875, -1.7905015 ]])
```

Observation

```
In [96]: sample_sigma=np.cov(s[0,:],s[1,:])
    ...: sample_mean=np.reshape(np.mean(s,1),(2,1))

In [97]: sample_mean
Out[97]:
array([[-0.07790852],
       [-0.29268621]])
```

Mean

```
In [98]: sample_sigma
Out[98]:
array([[0.63536345, 0.26843084],
       [0.26843084, 0.7267491 ]])
```

Covariance

```
In [99]: def md(t):
    ...:     return(np.linalg.inv(1+np.dot(np.dot((t-sample_mean).T,np.linalg.inv(sample_sigma)),(t-mu))))

In [100]: mds=np.zeros((1,n))
     ...: maxmd=0
     ...: maxi=0
     ...: for i in range(n):
     ...:     t=np.reshape(s[:,i],(2,1))
     ...:     mds[0,i]=md(t)
     ...:     if md(t)>maxmd:
     ...:         maxmd=md(t)
     ...:         maxi=i

In [101]: mds
Out[101]:
array([[0.6051441 , 0.4337282 , 0.61050523, 0.19414774, 0.42878172,
        0.30891884, 0.29229238, 1.00158896, 0.38884484, 0.21310353]])
```

Depth

```
In [102]: maxmd
Out[102]: array([[1.00158896]])

In [103]: maxi
Out[103]: 7

In [104]: s[:,7]
Out[104]: array([-0.02806256, -0.27307146])
```

The bivariate vector that maximizes the depth function b)

```
In [146]: n=1000
     ...: mu = np.array([[0, 0]]).T
     ...: Sigma = np.array([[1, 0.5], [0.5, 1]])
     ...: R = cholesky(Sigma)
     ...: s = np.dot(R,np.random.randn(2,n)) + mu
     ...: #plt.scatter(s[0,:],s[1,:])
     ...:
     ...:
     ...: sample_sigma=np.cov(s[0,:],s[1,:])
     ...: sample_mean=np.reshape(np.mean(s,1),(2,1))
     ...:
     ...: def md(t):
     ...:     return(np.linalg.inv(1+np.dot(np.dot((t-sample_mean).T,np.linalg.inv(sample_sigma)),(t-mu))))
     ...:
     ...:
     ...: mds=np.zeros((1,n))
     ...: maxmd=0
     ...: maxi=0
     ...: for i in range(n):
     ...:     t=np.reshape(s[:,i],(2,1))
     ...:     mds[0,i]=md(t)
     ...:     if md(t)>maxmd:
     ...:         maxmd=md(t)
     ...:         maxi=i
     ...:
     ...:
     ...: md_5=np.percentile(mds,5)
     ...: j=0
     ...: flag=np.zeros((2,50))
     ...: for i in range(n):
     ...:     if mds[0,i]<=md_5:
     ...:         flag[:,j]=s[:,i]
     ...:         j=j+1

     ...: plt.figure(figsize=(10,8))
     ...: plt.plot(s[0,:],s[1,:],'o',flag[0,:],flag[1,:],'ro')
Out[146]:
[<matplotlib.lines.Line2D at 0x1198bccb668>,
 <matplotlib.lines.Line2D at 0x1198bccb780>]
```