Simulation Home Work 3

Sekhar Mekala

September 27, 2016

1. Starting with $X_0 = 1$, write down the entire cycle for $X_i = 11X_i - 1$ mod (16).

Answer:

We can have a maximum of 16 unique numbers, since we are using $mod16$.

If we use $x_0 = 1$ as the seed, then we will get the following sequence of random numbers

```
rm(list=ls())

library(knitr)

#Create a vector of length 16
x <- vector(length=16)

#Initialize the first number to 1
x[1] <- 1

#Generate the remaining elements
for(i in 2:16)
   {
     x[i] = (11*x[i-1]) %% 16
}

#Create the data frame for display
#df <- data.frame(x=x)
df=data.frame(Random_Number=x)

#Display the data frame

kable(df)
```

Random_Number
1
11
9
3
1
11
9
3
1
11
9
3
1
11

9
3

As per the above display, after generating 4 numbers (1,11,9,3), the numbers repeat. Hence the entire cycle is $1, 11, 9, 3$

2. Using the LCG provided below: $X_i = (X_{i-1} + 12)mod(13)$, plot the pairs $(U_1, U_2), (U_2, U_3), .,$ and observe the lattice structure obtained. Discuss what you observed.

Answer:

Let us generate the sequence first, by using a seed say 100.

```
#Create a vector of length 14 (13 random elements, and the initial element for the seed).
x <- vector(length=14)

#Initialize the first element to 100. But you can pick any number as the seed.
x[1] <- 100

#Generate the random numbers
for(i in 2:14)
{
  x[i] <- (x[i-1] + 12) %% 13
}

#Display the generated random numbers
df <- data.frame(Random_Number=x[-1])
kable(df)
```

Random_Number
8
7
6
5
4
3
2
1
0
12
11
10
9

We used 100 as the seed to generate the above displayed random numbers. The random series begins at 8. This number will change if we use a different seed. The difference between the consecutive random numbers displayed above is 1, till the series reaches 0. After reaching 0, the series again begins at 12, and the consecutive random numbers have a difference of 1.

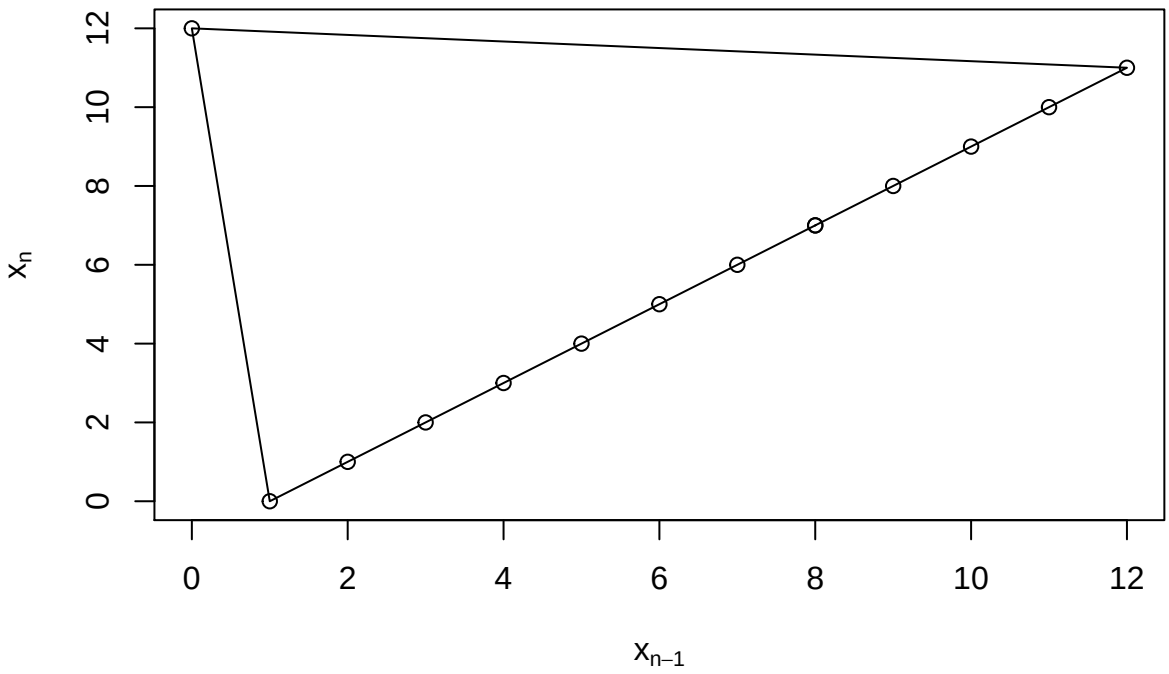
Let us plot the consecutive pairs (x1,x2)(x2,x3)...(x13,x14). Note that the (x13,x14) is the same as (x1,x2), since the cycle repeats after 13 numbers (since mod 13 is used):

```r
#Discard the seed
x <- x[-1]

#Initialize the y-coordinate
y <- x[-1]

y[13] <- x[1]

x[14] <- x[1]
y[14] <- y[1]


df <- data.frame(x_coordinate=x,y_coordinate=y)

kable(df)
```

x_coordinate	y_coordinate
8	7
7	6
6	5
5	4
4	3
3	2
2	1
1	0
0	12
12	11
11	10
10	9
9	8
8	7

```r
plot(x,y,xlab=expression(x[n-1]),
     ylab=expression(x[n]),main="Figure:2.1 Plot between the consecutive pairs of x")
lines(x,y)
```

Figure:2.1 Plot between the consecutive pairs of x



From the above display we can make the following conclusions:

- Based on the seed value, the initial random number is generated. This initial random number is always one of the numbers between 0 and 12. In the above example, we used seed as 100, and this had begun the numbers from 8. If we use 200 as the seed, then the first random number will be 4
- Except the first random number, the subsequent random numbers are always one less than the previous random number, till they reach 0, and, once they reach 0, the number 12 is generated, and the consecutive random numbers are decremented again by 1. This cycle is repeated.

3. Implement the pseudo-random number generator:

$$X_i = 16807X_{i-1} \text{mod}(2^{31} - 1)$$

Using the seed $X_0 = 1234567$, run the generator for 100,000 observations. Perform a chi-square goodness-of-fit test on the resulting PRN's. Use 20 equal-probability intervals and level $\alpha = 0.05$. Now perform a runs up-and-down test with $\alpha = 0.05$ on the observations to see if they are independent.

Answer

Let us calculate the chi-square value.

```r
#Create a vector of length 100000
x <- vector(length=100000)

#Initialize the first random number generated using the seed value 1234567
x[1] <- (1234567*16807)%%(2147483647)

#Generate the remaining random integers
for(i in 2:100000)
{
  x[i] = (16807*x[i-1])%%(2147483647)

}

#Divide by 2^31, to make the numbers belonging to U[0,1]
x <- x/2^31

#Create 20 intervals
intervals <- seq(from=0.05,to=1,by=0.05)

#Let us create O vector. This will contain the number of observations
#within each of the 20 intervals

O <- vector(length=20)

O[1] <- sum(x <= intervals[1])

for(i in 1:19)
{
  O[i+1] <- sum((intervals[i] < x & x <= intervals[i+1]))
}

#Let us create the expected value in each interval
E <- rep(100000/20,20)

#Getting O - E
O_E <- O - E

#Squaring the difference
O_E_sq <- (O_E)^2

#Dividing teh squared difference by E
O_E_sq_div_E <- O_E_sq/E

#preparing a data frame
df <- data.frame(O,E,O_E,O_E_sq,O_E_sq_div_E)
kable(df)
```

O	E	O_E	O_E_sq	O_E_sq_div_E
5066	5000	66	4356	0.8712
5030	5000	30	900	0.1800
5045	5000	45	2025	0.4050
5087	5000	87	7569	1.5138
4947	5000	-53	2809	0.5618

O	E	O_E	O_E_sq	O_E_sq_div_E
4954	5000	-46	2116	0.4232
4936	5000	-64	4096	0.8192
4933	5000	-67	4489	0.8978
4900	5000	-100	10000	2.0000
4958	5000	-42	1764	0.3528
5087	5000	87	7569	1.5138
4995	5000	-5	25	0.0050
5076	5000	76	5776	1.1552
5019	5000	19	361	0.0722
5003	5000	3	9	0.0018
5067	5000	67	4489	0.8978
4980	5000	-20	400	0.0800
4919	5000	-81	6561	1.3122
5058	5000	58	3364	0.6728
4940	5000	-60	3600	0.7200

Let us state the null and alternate hypothesis:

$$H_0: \ R \sim Uniform[0,1]$$
$$H_a: \ R \nsim Uniform[0,1]$$

We will get the χ^2 value as 14.4556. At 19 degrees of freedom, the p-value associated with this value is 0.756514, which is way higher than the desired significance level, 0.05. Hence, we are unable to reject the null hypothesis. Therefore, the numbers are uniformly distributed.

Runs up-and-down test of independence

In Runs up-and-down test we get the difference between the consecutive random numbers, and whenever there is a change in the sign of the resultant consecutive numbers difference, we count the change as a run. If a is the total number of runs in a truly random sequence, the mean and variance of a is given by

$$\mu_a = \frac{2N-1}{3}$$
$$\sigma_a^2 = \frac{16N-29}{90}$$

Where N = number of observations, and if $N > 20$ then the distribution of the runs can be approximated by normal distribution with the mean μ_a and variance σ_a^2 (computed using the above formulae).

Since N = 100000 for our data, we can compute $\mu_a = \frac{200000-1}{3} = 66666.33$ and variance, $\sigma_a^2 = \frac{1600000-29}{90} = 17777.46$

Let us obtain the runs of the generated random data.

```
#Get the difference between the the consecutive random numbers
y <- x[-1] - x[-length(x)]

#Initialize the runs counter
a <- 0

#Check how often the sign changes
```

```
for(i in 2:length(y))
{
  #Increment the counter only when the sign changes
  if((y[i-1] < 0 & y[i] >= 0) |(y[i-1] >= 0 & y[i] < 0) )
  {
    a <- a+1
  }

}


#Refer to the below website to know about the run up-down testing
#http://www.eg.bucknell.edu/~xmeng/Course/CS6337/Note/master/node44.html

#Find the p-value
mu <- (2*100000-1)/3
sd <- sqrt((16*100000 - 29)/90)

#pnorm((r - mu)/sd)
```

Let us state the following hypothesis:

$$H_0: \ R \sim \text{are independent}$$
$$H_a: \ R \nsim \text{are independent}$$

We obtained 66734 runs, and the p-value associated with this value is 0.6940996. The confidence interval at 0.05 significance level is $[66405.01, 66927.66]$. Since the runs obtained lies within the confidence interval range, we are unable to reject the null hypothesis, and hence the generated random numbers are independent.

4. Give inverse-transforms, composition, and acceptance-rejection algorithms for generating from the following density:

$$f(x) = \begin{cases} \frac{3x^2}{2}, -1 \le x \le 1 \\ 0, \quad otherwise \end{cases}$$

Answer

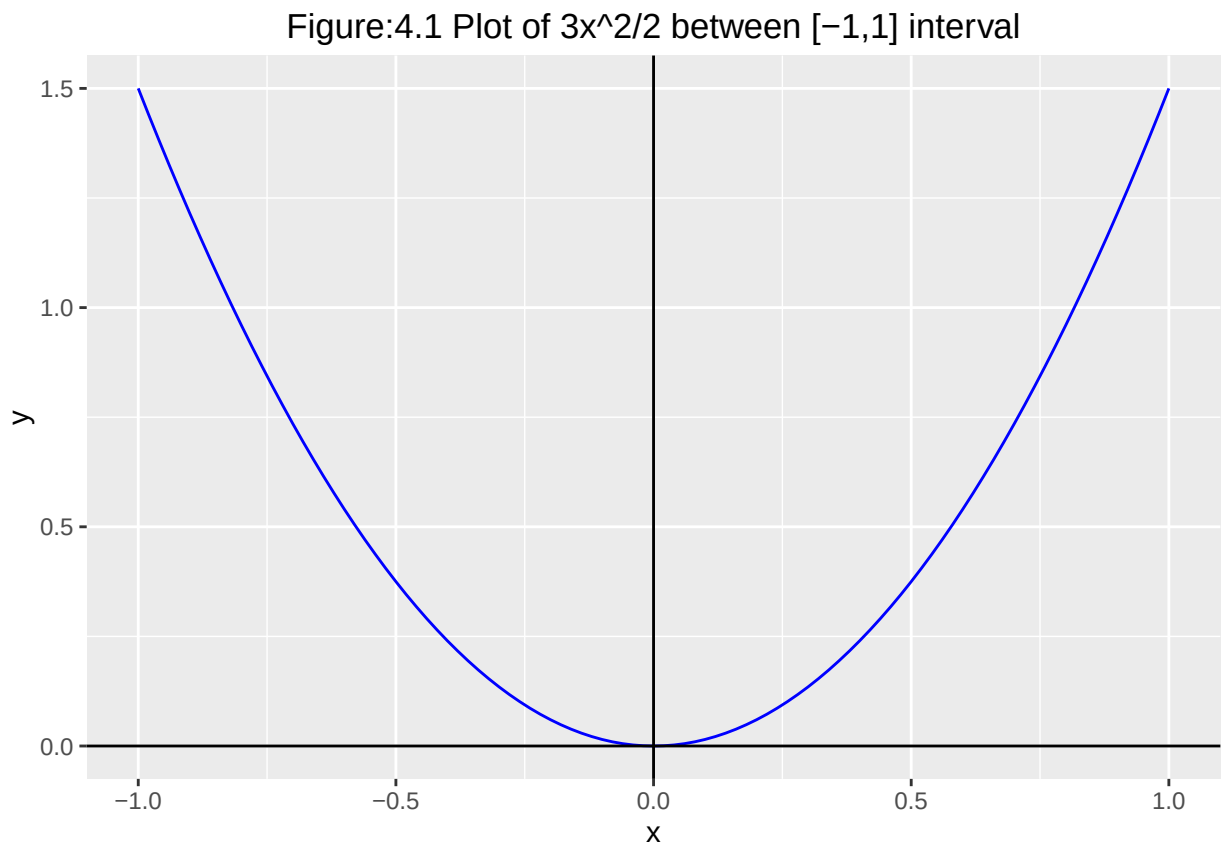
Given that the PDF as $f(x) = \frac{3x^2}{2}$, when $-1 \le x \le 1$

Inverse-transform method

Let us draw the PDF between the intervals $[-1, 1]$

```
library(ggplot2)

myfun <- function(x)
{
  3*x^2/2
```

```
}

ggplot(data.frame(x=c(-1,1)),aes(x))+
  stat_function(fun=myfun,color="blue")+
  geom_vline(xintercept = 0)+
  geom_hline(yintercept = 0)+
  labs(title="Figure:4.1 Plot of 3x^2/2 between [-1,1] interval")
```



The function is symmetrical on the y-axis. So we will use the CDF between the interval $[0, 1]$ and use the inverse transform method to obtain a random number, r, between $[0, 1]$ for the PMF function $3x^2/2$, and generate another random number (say y), and if y<=0.5, then we return r, else we return -r. This way we can generate the random numbers for the PMF $3x^2/2$ between $[-1, 1]$.

Let us obtain CDF between $[0, 1]$:

$$\int_0^x 3y^2/2dy = x^3/2$$

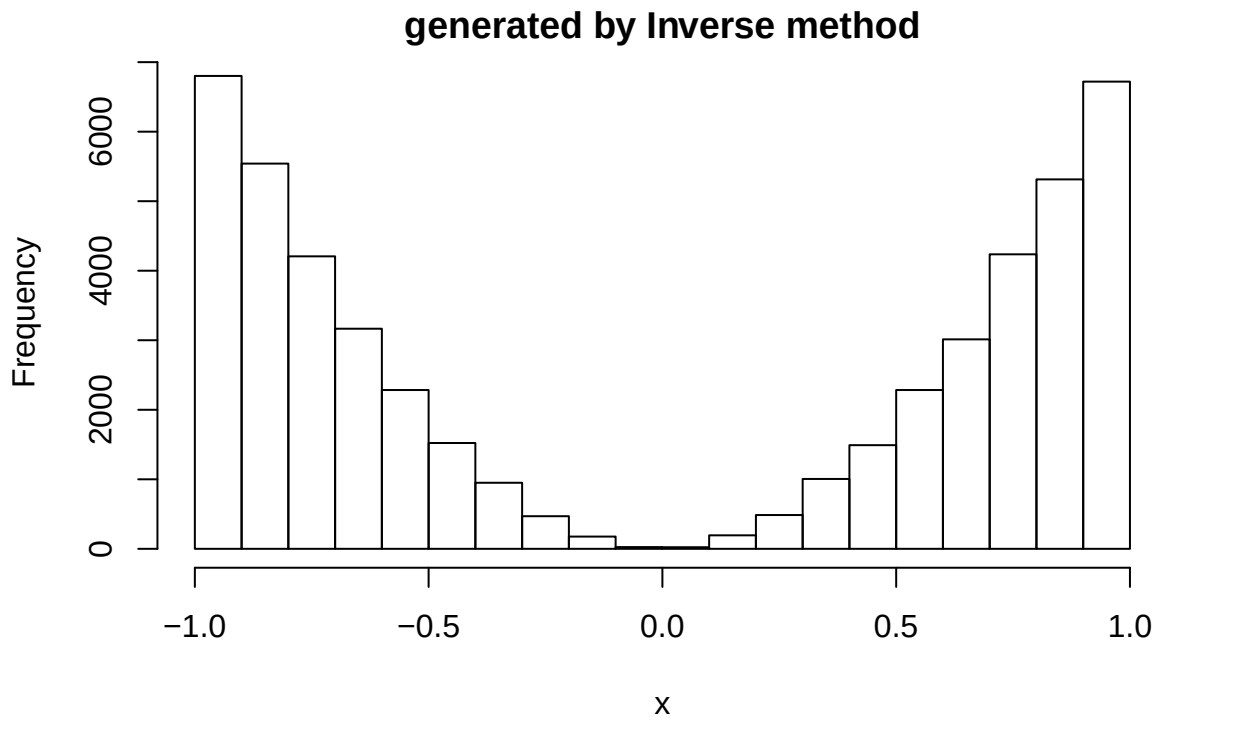
The above obtained CDF will be equated to a random number R, generated using the uniform distribution between the interval $[0, 1]$.

$$x^3/2 = R$$
$$x = (2R)^{1/3}$$

Now, we will generate another random number y having uniform distribution between $[0, 1]$. If y <= 0.5, then we return x, else -x

The following R code will implement this logic:

```
my_rand <- function(){
  #Generate a uniform Random number

  R <- runif(1,0,1)

  #Getting the value of x
  x <- (2 * R)^(1/3)

  #Decide the sign
  x <- ifelse(runif(1,0,1) <=0.5,x,-x)
  return(x)
}

#Generating 100000 samples
x <- replicate(100000,my_rand())
x <- x[x<=1 & x>= -1]
hist(x,main="Figure:4.2 Histogram of random numbers belonging to 3x^2/2 PMF,
     \ngenerated by Inverse method")
```



The figure 4.2 matches the distribution of the function $3x^2/2$ displayed in figure 4.1

Composition method:

The interval $[-1, 1]$ can be divided into $[-1, 0]$ and $[0, -1]$, and obtain the two CDF functions:

CDF-1 between the interval $[-1, 0]$

$$\int_x^0 3y^2/2dy = \frac{-x^3}{2}$$

The inverse of CDF-1 will be:

$$x = (-2R)^{(1/3)} = -(2R)^{(1/3)}, \text{ when you ignore the complex numbers}$$

CDF-2 between the interval $[0, 1]$

$$\int_0^x 3y^2/2dy = \frac{x^3}{2}$$

The inverse of CDF-2 will be:

$$x = (2R)^{(1/3)}$$

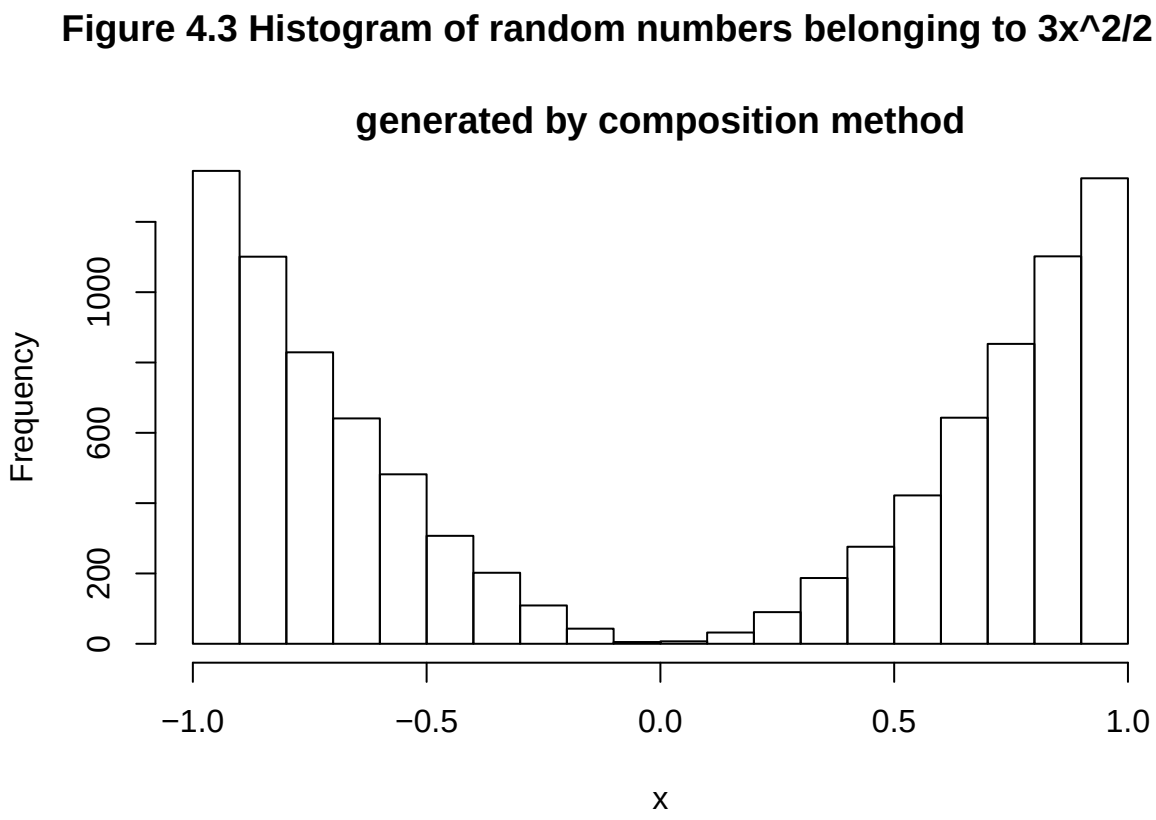
We will generate 2 uniform random numbers R_1 and R_2, and if R_1 is less than or equal to 0.5, then we will use inverse CDF-1 function to get a random number using R_2, else we will use the inverse CDF-2 function to get a random number using R_2. In this generation process, if we get any number outside $[-1, 1]$, then we will ignore that random number.

The following R code will implement this logic:

```
rand <- function()
{

repeat{
  #Generate 2 random numbers from Unif dist.
  R1 <- runif(1)
  R2 <- runif(1)


if(R1 <= 0.5){
  x <- (-(2*R2)^(1/3))
}
else
  {
  x <- ((2*R2)^(1/3))
  }
if(x >= -1 & x <= 1)
  {return(x)}
}

}
```

Let us generate some sample values using the R code (based on the composition method), given above:

```
x <- replicate(10000,rand())
hist(x,main="Figure 4.3 Histogram of random numbers belonging to 3x^2/2 PMF,
     \ngenerated by composition method")
```



The figure 4.3 matches the distribution of the function $3x^2/2$ displayed in figure 4.1

Accept-Reject method

For the PDF $f(x) = 3x^2/2$, in the interval [-1,1], we can say that another function $g(x) = 3/2$ is strictly greater than or equal to the given pdf. We can write the following expression:

We will generate two random numbers R_1 and R_2 from the uniform distribution, $U[0, 1]$, and if

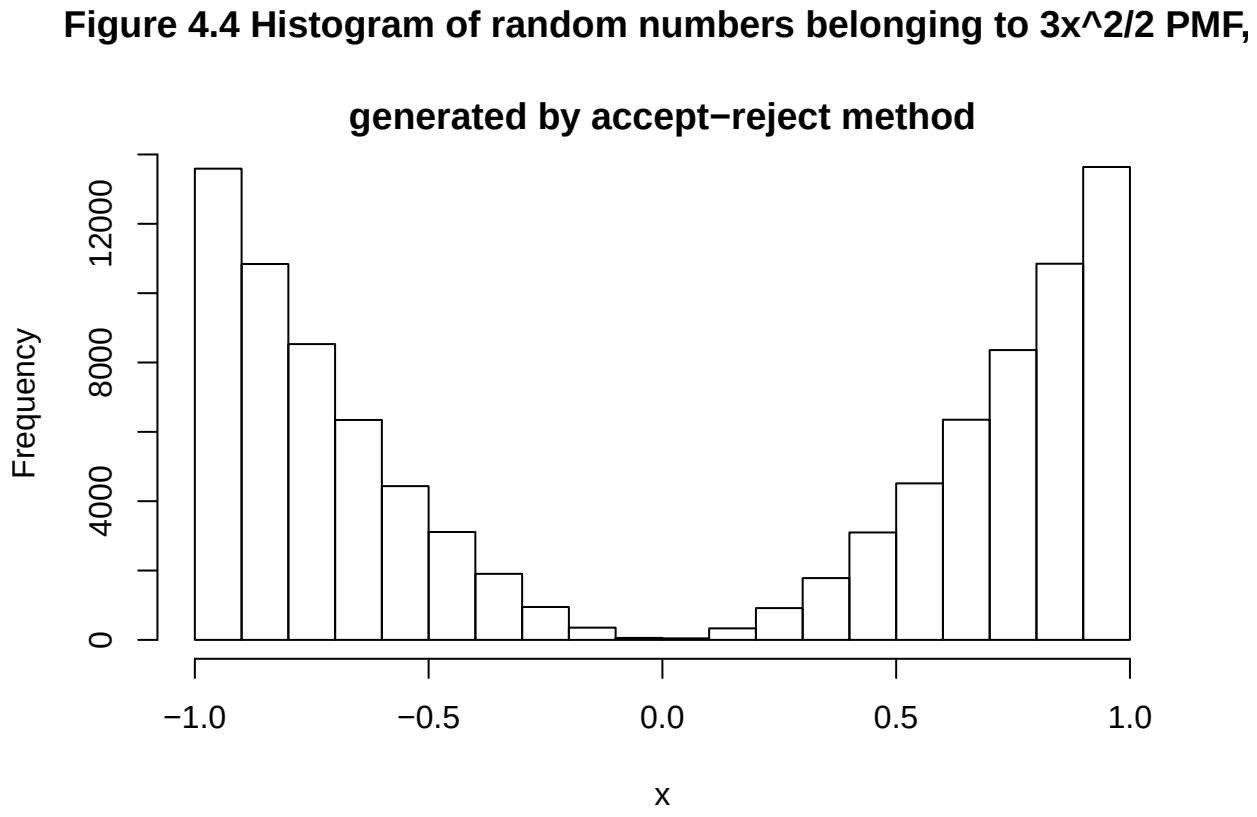
$$R1 \leq \frac{f(R_2)}{g(R_2)}$$

, then return R_2 as the random number, else repeat the procedure. Given the symmetric nature of the function $f(x) = 3x^2/2$, we will return positive random number and negative random numbers with equal probability.

$$\frac{f(x)}{g(x)} = \frac{3x^2/2}{3/2} = x^2$$

Hence, if $R_1 \leq R_2^2$, then return R_2 as the random number, else repeat. The sign or R_2 is either positive or negative with equal probability. The following R code implements the discussed logic of accept-reject method:

```
rand <- function()
{
repeat{
R1 <- runif(1)
R2 <- runif(1)

if(R1 <= R2^2)
{x <- R2
  break}
}
  x <- ifelse(runif(1)<= 0.5,x,-x)
  return(x)

}

x <- replicate(100000,rand())
hist(x,main="Figure 4.4 Histogram of random numbers belonging to 3x^2/2 PMF,
     \ngenerated by accept-reject method")
```



The figure 4.4 matches the distribution of the function $3x^2/2$ displayed in figure 4.1

Problem-5a

The inverse of a normal CDF is not easy to compute, but we can use an approximate of the inverse of the normal CDF function (as discussed in http://www.m-hikari.com/ams/ams-password-2008/ams-password9-12-2008/

alodatAMS9-12-2008-2). We will use the following approximation to normal CDF:

$$x = \sqrt{-\sqrt{\frac{8}{\pi}log(1-(p-0.5)^2)}}, \text{ where p = random number from U[0,1]}$$

We will use the above formula to generate random numbers which are normally distributed. The sign (+/-) of the random number is also decided randomly with equal probability.

The following R code will perform this:

```
rm(list=ls())

library(knitr)
library(ggplot2)
library(pander)
library(reshape2)

normrandit <- function()
{
  #Generating 1 uniform random number
  p <- runif(1)
  plus_or_minus <- sample(x=c(-1,1),1)
  return(plus_or_minus * sqrt(-log(1-(2*p -1)^2)/sqrt(pi/8)))
}

#Creating another function itstats()

itstats <- function(N)
{
  x <- replicate(N,normrandit())
  return(list(mean=mean(x),std_dev = sd(x)))

}

#Calling itstats() function with 100000 as the input.
itstats(100000)
```

```
## $mean
## [1] 8.71129e-05
##
## $std_dev
## [1] 0.9900907
```

The generated mean is approximately 0, and the standard deviation is approximately 1.

Problem-5b

Let us use the Box-Muller algorithm to generate the random numbers. The function normrandbm() is defined to implement the Box-Muller method:

```
normrandbm <- function()
{
U <- runif(1)
V <- runif(1)
X <- sqrt(-2*log(U)) * cos(2*pi*V)
Y <- sqrt(-2*log(U)) * sin(2*pi*V)
return(c(X,Y))
}

bmstats <- function(N)
{
  x <- replicate(N,normrandbm())
  return(list(mean=mean(x),std_dev=sd(x)))
}

bmstats(100000)
```

```
## $mean
## [1] -0.0003280366
##
## $std_dev
## [1] 0.9990509
```

When the function bmstats() is ran with 100000 as the input, it produced approximately a 0 mean and approximately 1 as the standard deviation.

5C. Generation of random numbers from normal distribution using accept-reject approach

The normrandr() function generates a random number from std. normal distribution using accept-reject method.

```
normrandar <- function()
{
  repeat{
  U1 <- runif(1)
  U2 <- runif(1)

  X <- -log(U1)
  Y <- -log(U2)

  if(Y >= (X-1)^2/2){
    sign <- sample(c(-1,1),1)
    return(sign*X)
  }
  }
}

arstats <- function(N)
{
  x <- replicate(N,normrandar())
  return(list(mean=mean(x),std_dev=sd(x)))
```

```
}

arstats(100000)
```

```
## $mean
## [1] 0.001082095
##
## $std_dev
## [1] 0.9987407
```

When the function arstats() is executed with 100000 it has returned a std. dev of 1 (approximately), and a mean of 0 (approximately).

Problem 5D.

Let us test the three functions created with various sample sizes:

```
N= c(100,1000, 10000,100000)

#Set the seed to 100
set.seed(100)

df_print <- data.frame()
for(j in 1:length(N))
  {

  mean_1 <-vector()

  mean_2 <-vector()

  mean_3 <-vector()

  sd_1 <-vector()

  sd_2 <-vector()

  sd_3 <-vector()

  t1 <- vector()
  t2 <- vector()
  t3 <- vector()


        for(i in 1:10)
        {
        t <- system.time(L <- itstats(N[j]))
        t1[i] <- t[2]
        mean_1 <- c(mean_1,L$mean)
        sd_1 <- c(sd_1,L$std_dev)

        t <- system.time(L<-bmstats(N[j]))
        t2[i] <- t[2]
```

```
        mean_2 <- c(mean_2,L$mean)
        sd_2 <- c(sd_2,L$std_dev)

        t <- system.time(L<-arstats(N[j]))
        t3[i] <- t[2]
        mean_3 <- c(mean_3,L$mean)
        sd_3 <- c(sd_3,L$std_dev)

        }

  df_print <- rbind(df_print,data.frame(N=N[j],
                                        itstats_mean=mean(mean_1),
                                        itstats_sd=mean(sd_1),
                                        itstats_CPU = mean(t1),
                                        bmstats_mean=mean(mean_2),
                                        bmstats_sd=mean(sd_2),
                                        bmstats_CPU = mean(t2),
                                        arstats_mean=mean(mean_3),
                                        arstats_sd=mean(sd_3),
                                        arstats_CPU = mean(t3)
                                      ))

  }

library(pander)
pander(df_print, split.table = 120,
       style = 'rmarkdown',
       caption="Performance of three methods")
```

Table 3: Performance of three methods (continued below)

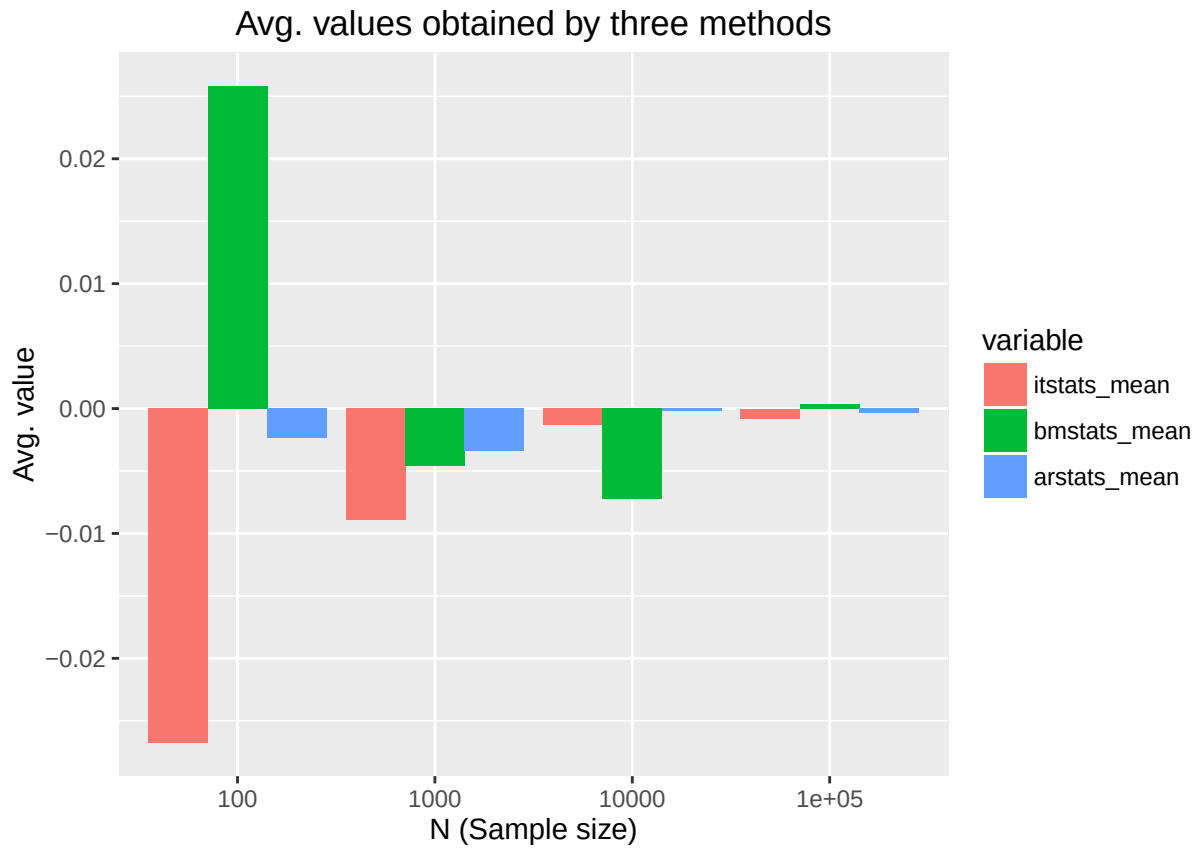
N	itstats_mean	itstats_sd	itstats_CPU	bmstats_mean	bmstats_sd	bmstats_CPU
100	-0.0268	0.9988	0	0.02586	1.006	0
1000	-0.008902	0.9835	0	-0.004609	1.004	0
10000	-0.001293	0.9867	0.002	-0.007273	0.9963	0
1e+05	-0.0007933	0.9903	0.006	0.0003955	0.9992	0.007

arstats_mean	arstats_sd	arstats_CPU
-0.002373	0.9655	0
-0.00341	1.003	0
-0.0001699	1.002	0
-0.0003826	0.9997	0.001

Let us plot the bar chart for the mean obtained for three methods at various values of N.

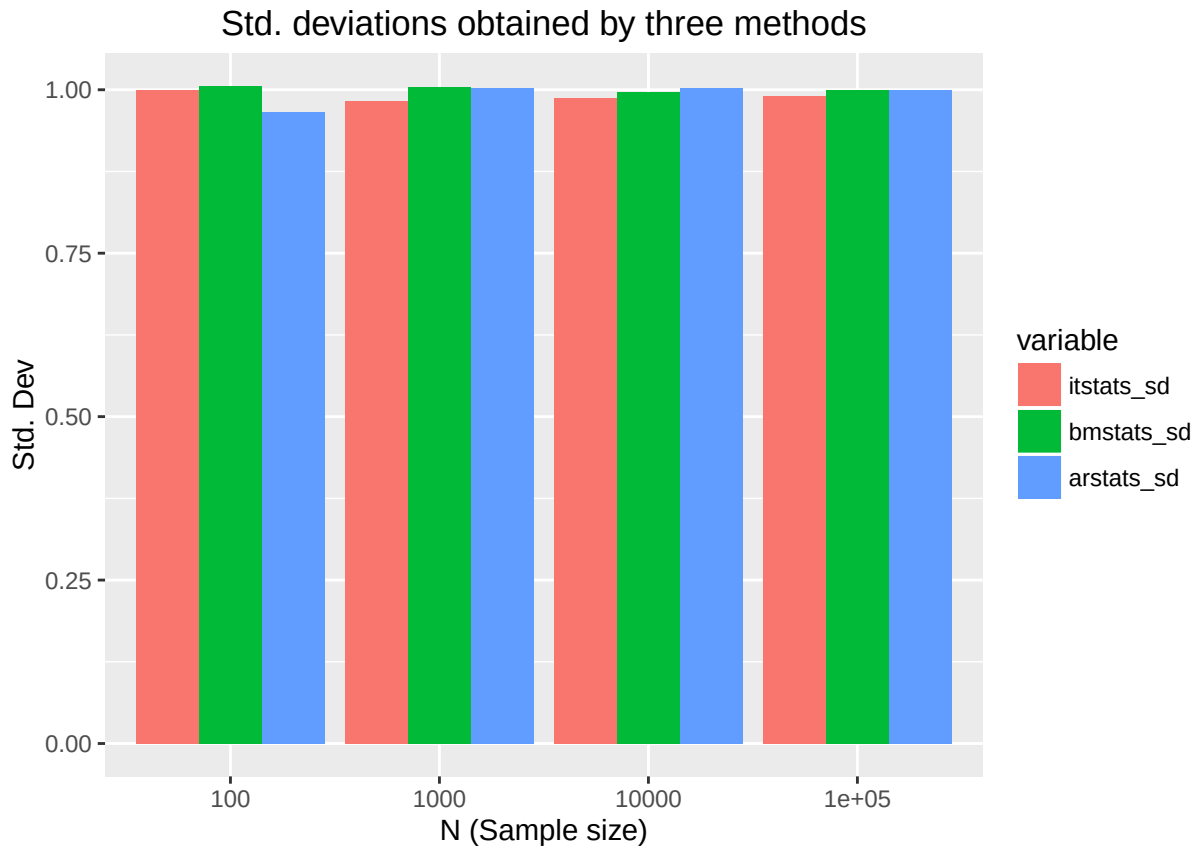
```
library(reshape2)

df_print$N <- factor(df_print$N)
df <- melt(df_print[,c("N","itstats_mean","bmstats_mean","arstats_mean")],id.vars="N")
```

```
library(ggplot2)

ggplot(df,aes(N,value,fill=variable))+
  geom_bar(stat="identity",position="dodge")+
  labs(title="Avg. values obtained by three methods",x="N (Sample size)",y="Avg. value")
```



Let us plot the bar chart for the standard deviations obtained for three methods at various values of N.

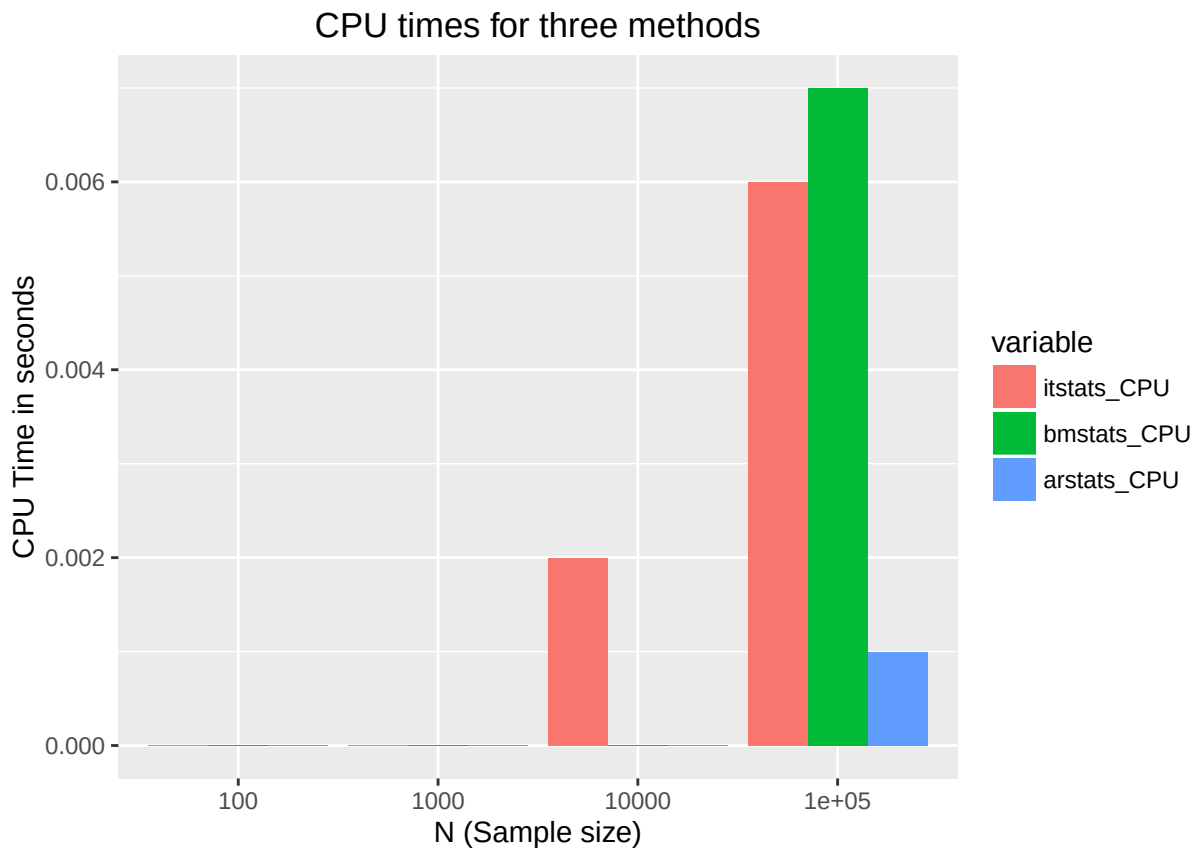
```
df <- melt(df_print[,c("N","itstats_sd","bmstats_sd","arstats_sd")],id.vars="N")

ggplot(df,aes(N,value,fill=variable))+
  geom_bar(stat="identity",position="dodge")+
  labs(title="Std. deviations obtained by three methods",x="N (Sample size)",y="Std. Dev")
```



The plots show that as the number of samples generated increase, the mean value approaches 0, and std. dev. reaches to 1. But of all the three methods, the Accept/Reject method performed the best, since the mean value obtained by this method is always closer to 0, when compared to other methods, for all the tested sample sizes.

Let us plot the CPU time for the three methods, at various sample sizes:

```
df <- melt(df_print[,c("N","itstats_CPU","bmstats_CPU","arstats_CPU")],id.vars="N")

ggplot(df,aes(N,value,fill=variable))+
  geom_bar(stat="identity",position="dodge")+
  labs(title="CPU times for three methods",x="N (Sample size)",y="CPU Time in seconds")
```

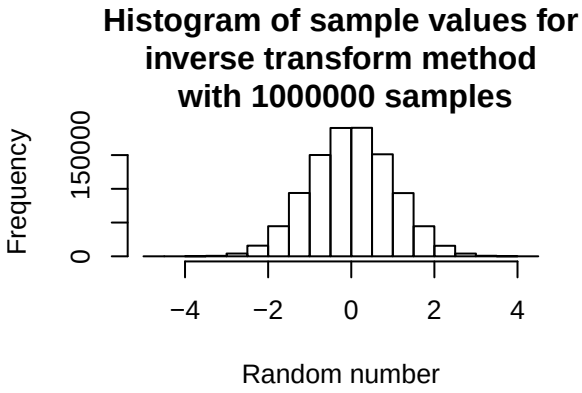


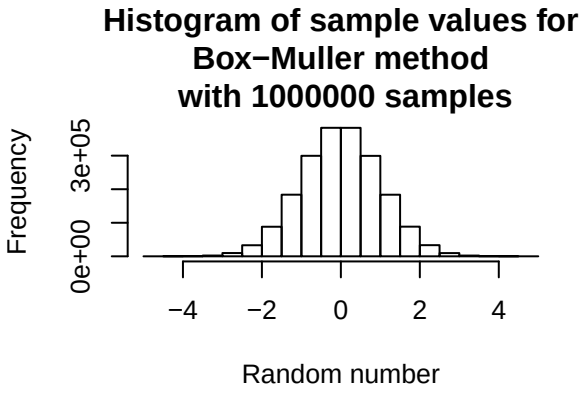
The CPU time increases as the number of samples increase. But I was getting different CPU times for the three methods for a sample size of 100000, for different runs. And since the CPU times cannot be controlled by a seed, I am not offering any further explanation about the CPU times. From the accuracy point of view, Accept-Reject method has the greater level of accuracy even for smaller sample sizes. But, as the sample sizes increase, the accuracy of all the methods have increased.

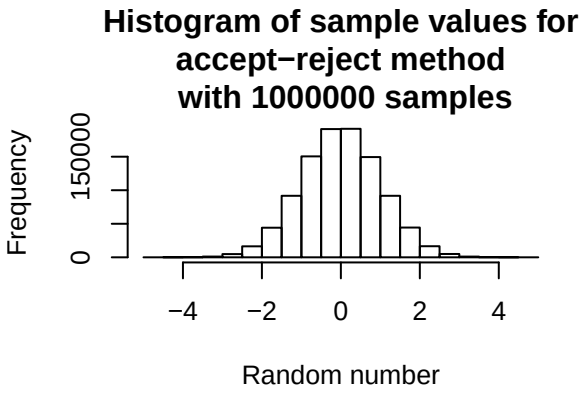
Problem 5E.

Let us plot the histograms for the three methods, with sample size of 1000000

```
par(mfrow=c(2,2))

hist(replicate(1000000,normrandit()),
     main="Histogram of sample values for\n inverse transform method \n with 1000000 samples",
     xlab="Random number")

hist(replicate(1000000,normrandbm()),
     main="Histogram of sample values for\n Box-Muller method \n with 1000000 samples",
     xlab="Random number")

hist(replicate(1000000,normrandar()),
     main="Histogram of sample values for\n accept-reject method \n with 1000000 samples",
     xlab="Random number")

par(mfrow=c(1,1))
```







All the three methods have generated almost identical distributions, which are identical to normal distribution.

Problem 6

Let use create "insidecircle()" that takes two inputs between 0 and 1 and returns 1 if these points fall within the unit circle.

```
insidecircle <- function(U1,U2)
  {
    ifelse(sqrt(U1^2+U2^2) <= 1, 1, 0)

  }
```

Let us create "estimatepi()" function that takes N as input, and generates N pairs of random numbers having Uniform distribution, and calculates π based on the number of points that fall within the quarter circle, inscribed in a unit square. Let us assume that the proportion of points that fall within the quarter circle as p and the points that do not fall within the circle will be $1 - p$. Therefore the std. error can be found as:

$$SE = \sqrt{\frac{p.(1-p)}{N}}$$

```
estimatepi <- function(N)
  {
    U1 <- runif(N)
    U2 <- runif(N)
```

```
    points <- insidecircle(U1,U2)

    sqrt(mean(points)*(1-mean(points)))
    sd(points)/sqrt(N)

    p <- mean(points)
    se <- sqrt(p*(1-p)/N)
    pi <- 4*p

    max <- pi + 1.96*se
    min <- pi - 1.96*se
    return(list(pi=pi,std_error=se,CI_min=min,CI_max=max))
  }
```

Problem 6C

Let us run the estimatepi() function for N=1000 to 10000 in increments of 500

```
set.seed(10)
cnt <- seq(from=1000,to=10000,by=500)


df <- data.frame()

for(i in 1:length(cnt))
  {
    l <- estimatepi(cnt[i])

    df <- rbind(df,data.frame(N=cnt[i],pi=l$pi,std_error=l$std_error,CI_min=l$CI_min,CI_max=l$CI_max))


  }

kable(df)
```

N	pi	std_error	CI_min	CI_max
1000	3.192000	0.0126963	3.167115	3.216885
1500	3.146667	0.0105774	3.125935	3.167398
2000	3.104000	0.0093227	3.085728	3.122272
2500	3.174400	0.0080944	3.158535	3.190265
3000	3.180000	0.0073705	3.165554	3.194446
3500	3.169143	0.0068571	3.155703	3.182583
4000	3.172000	0.0064061	3.159444	3.184556
4500	3.141333	0.0061207	3.129337	3.153330
5000	3.120000	0.0058583	3.108518	3.131482
5500	3.177455	0.0054498	3.166773	3.188136
6000	3.180000	0.0052118	3.169785	3.190215
6500	3.168615	0.0050329	3.158751	3.178480
7000	3.164000	0.0048597	3.154475	3.173525
7500	3.149867	0.0047239	3.140608	3.159126
8000	3.148000	0.0045775	3.139028	3.156972

N	pi	std_error	CI_min	CI_max
8500	3.133176	0.0044688	3.124418	3.141935
9000	3.144889	0.0043215	3.136419	3.153359
9500	3.157474	0.0041835	3.149274	3.165673
10000	3.136800	0.0041138	3.128737	3.144863

Let us find for what sample size does the π value estimated is within the range of 0.1 from the true π value.

```
min_sample <- cnt[which(abs((df$pi - pi))<=0.1)[1]]
```

Therefore with 1000 sample size we can get an estimate of π, that is within the ± 0.1 range.

Problem 6d

Using the value of N = 1000, let us run the estimatepi() for 500 times and collect 500 estimates of π

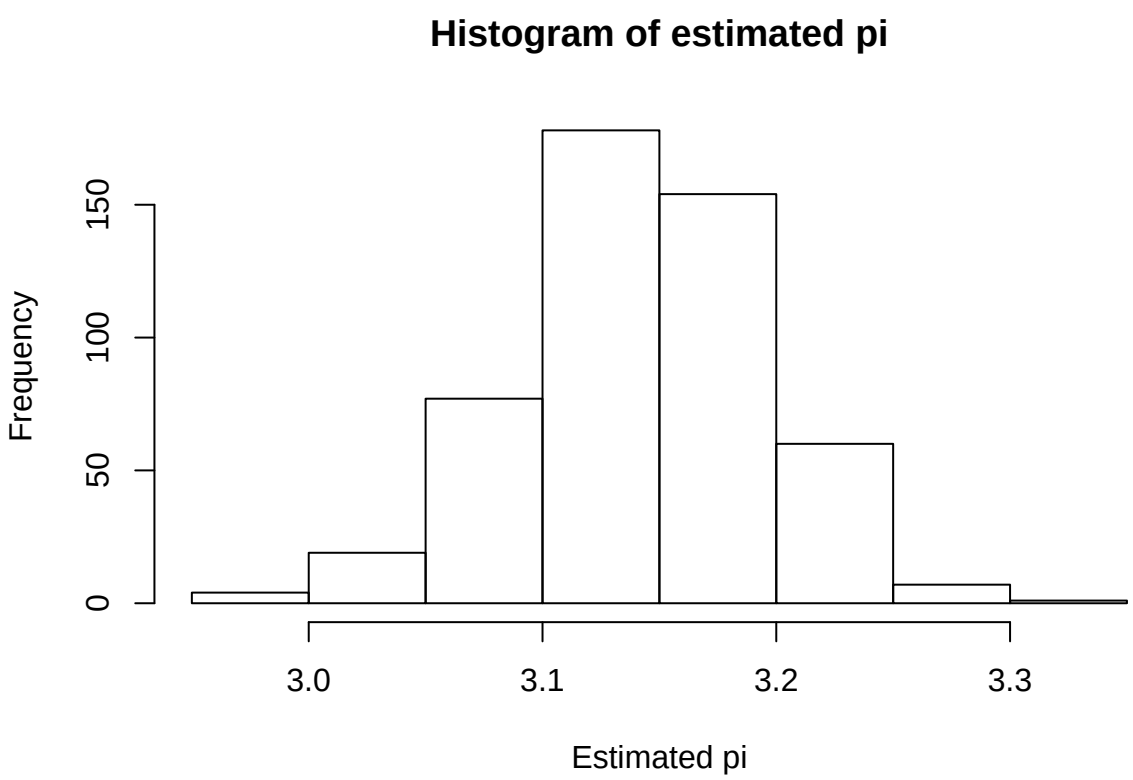
```
set.seed(10)
e_pi <- replicate(500,estimatepi(1000)$pi)
par(mfrow=c(1,1))
mean(e_pi)
```

```
## [1] 3.1432
```

```
sd(e_pi)
```

```
## [1] 0.05383922
```

```
hist(e_pi,main="Histogram of estimated pi",xlab="Estimated pi")
```



The π value obtained is 3.1432 and std error is 0.0538392. In problem 6c, we obtained a std. error of 0.0126963. The std error 0.0538392 obtained here is different from the standard error obtained in problem 6c for a sample size of 1000.

The percentage of samples obtained in this problem that are within the control interval range obtained for the sample size of 1000 is 26%.