IS604_charleyferrari_hw3

Charley Ferrari

Wednesday, September 30, 2015

Question 1

Starting with $X_0 = 1$, write down the entire cycle for

$$X_i = 11X_{i-1} mod(16)$$

```
x <- 1

for(i in 1:7){
  x <- c(x,(11*tail(x,1)) %% 16)
}

x
```

```
## [1]  1 11  9  3  1 11  9  3
```

This cycle repeats every 4 values: 1, 11, 9, 3.

Question 2

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

The seed won't matter here, because this method of random number generation will cycle in the same predictable way no matter where you start:

```
x <- 0

for(i in 1:25){
  x <- c(x,(tail(x,1) + 12) %% 13)
}

x
```

```
##  [1]  0 12 11 10  9  8  7  6  5  4  3  2  1  0 12 11 10  9  8  7  6  5  4
## [24]  3  2  1
```

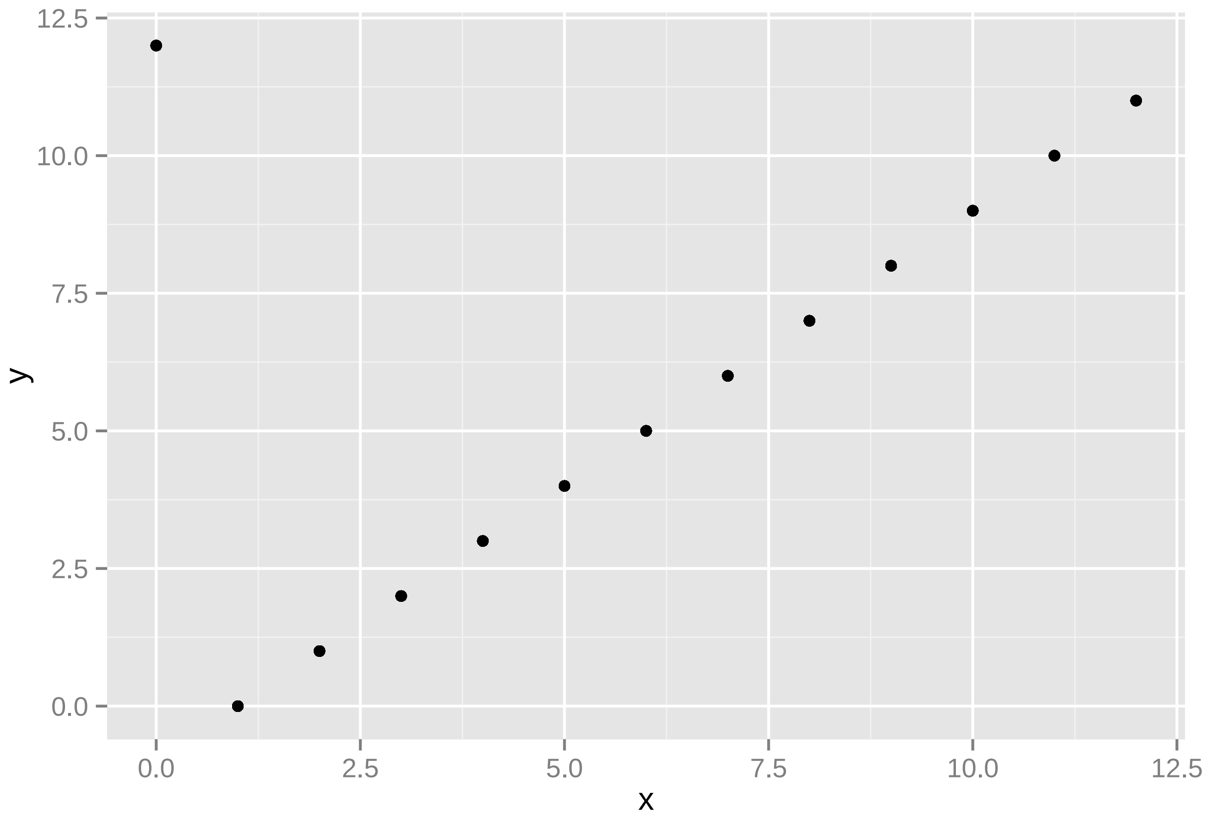
```
x <- 3

for(i in 1:25){
  x <- c(x,(tail(x,1) + 12) %% 13)
}

x
```

```
##  [1]  3  2  1  0 12 11 10  9  8  7  6  5  4  3  2  1  0 12 11 10  9  8  7
## [24]  6  5  4
```

So, wherever the seed is, the sequence will start decreasing to 0, then reset back to 12 and start decreasing again back to 0.

Lets plot these pairs out:

```
x <- 0

for(i in 1:1000){
  x <- c(x,(tail(x,1) + 12) %% 13)
}

library(ggplot2)

data <- data.frame(x=head(x,-1), y=tail(x, -1))

ggplot(data,aes(x=x,y=y)) + geom_point()
```



As expected by the cycling of the variables, a very defined pattern emerges. The pairs repeat, just as the x values repeat. It displays as a line, with a single point at (0,12)

Question 3

Implement the Pseudo-random number generator:

$$X_i = 16807 X_{i-1} mod(2^3 1 - 1)$$

Using the seed $X_0 = 1234567$, run the generator for 100,000 observations. Perform a chi-squared goodness of fit test on the resulting PRNs. 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.

```r
x <- 1234567
r <- 1234567/(2^31 - 1)

for(i in 1:9999){
  x <- c(x, (16807*tail(x,1)) %% (2^31 - 1))
  r <- c(r, tail(x,1)/(2^31 - 1))
}

classes <- seq(0,1,by=1/20)

table <- data.frame(lower=head(classes,-1),upper=tail(classes,-1))

library(plyr)

loweruppercount <- function(data){
  c(intervalcount = with(data, length(r[r>=lower&r<upper])))
}

chitable <- ddply(table, .variables=c("lower","upper"), .fun=loweruppercount)

chitable$chistat <- ((chitable$intervalcount - length(r)/20)^2)/(length(r)/20)

chisquare <- sum(chitable$chistat)

chisquare
```

```
## [1] 14.636
```

with n=20 (19 degrees of freedom), the critical value of chi square is 30.1. The value calculated above is smaller than this value, so the null hypothesis of a uniform distribution is not rejected.

runs up and down test:

```r
runs <- ifelse(head(r,-1)<tail(r,-1),1,-1)
```

I'll count the runs by counting how many times R_i in the runs vector above is different from R_{i-1}. if these two values are different, a run has occurred. If not, a run is still in progress.

```r
S <- sum(head(runs,-1)!=tail(runs,-1))

S
```

```
## [1] 6668
```

E(S) should be $\frac{2N-1}{3}$. Lets find out our the calculated S compares:

```r
Es <- (2*length(r)-1)/3

Es
```

```
## [1] 6666.333
```

```r
Se <- sqrt((16*length(r)-29)/90)

ZS <- (S-Es)/Se

ZS
```

```
## [1] 0.03953205
```

This is within the boundary of 1.96 95% confidence interval, so we can accept the hypothesis that this data is uniformly distributed.

Question 4

Give Inverse Transforms, composition, and acceptance-rejection algorithms for generating from the following density:

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

Inverse Transform:

Our region of interest is [-1,1]. The density function is 0 outside of this region, meaning there is 0 probability that the random variable can fall outside of this region. As with all CDFs, the CDF of this random variable will increase from 0 to 1. In the range $-\infty$ to -1, the CDF will be 0. From -1 to 1, the CDF will increase from 0 to 1, and from 1 to ∞ the CDF will remain constant at 1.

So, the CDF will be defined as:

$$F(x) = 0, -\infty < x < -1$$
$$F(x) = \int_{-1}^{x} \frac{3x^2}{2}, -1 \le x \le 1$$
$$F(x) = 1, 1 < x < \infty$$

$$\int_{-1}^{x} \frac{3x^2}{2} = \frac{x^3}{2} - \frac{-1^3}{2} = \frac{x^3+1}{2}$$

Which means our CDF is:

$$F(x) = 0, -\infty < x < -1$$
$$F(x) = \frac{x^3+1}{2}$$
$$F(x) = 1, 1 < x < \infty$$

Now that I have the CDF, I can set $y = F(x)$, then solve for x to get $F^{-1}(x)$. This function will be undefined outside of $0 \leq x \leq 1$. So, we have:

$$y = \frac{x^3 + 1}{2}$$

$$x = (2y - 1)^{\frac{1}{3}}$$

So, our inverse CDF is defined as:

$$F^{-1}(x) = (2x - 1)^{\frac{1}{3}}, 0 \leq x \leq 1$$
$$F^{-1}(x) = undefined, otherwise$$

So, in order to sample from a random variable with this distribution, we can sample from a uniform distribution between 0 and 1, and then run each of those samples through the $F^-1(x)$ we calculated above.

First, I want to generate a set of random numbers from -1 to 1. I'll use runif to generate my sample:

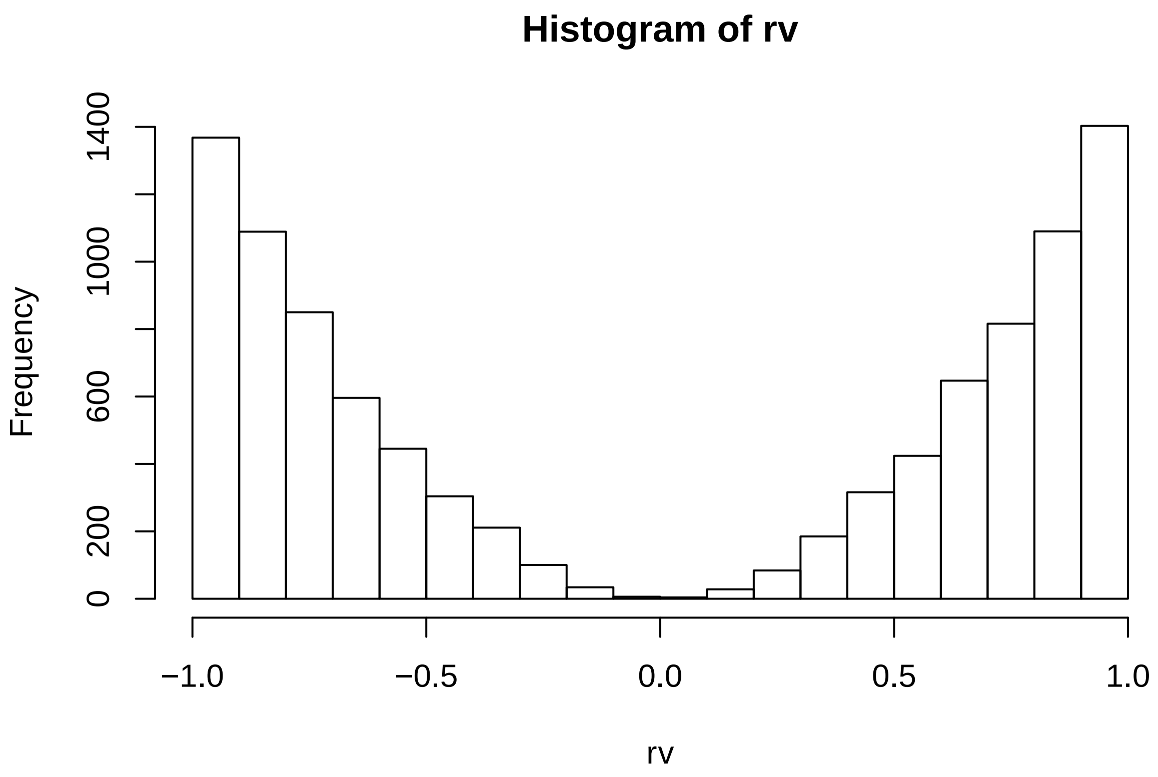
```
x <- runif(10000)
```

I'm going to have to define my own cube root function, since R doesn't accept the real cube roots of negative numbers, and instead gives me NaN. I'll define it below:

```
cuberoot <- function(x){
  return(sign(x)*(abs(x)^(1/3)))
}
```

Now I can use this to generate my random variable:

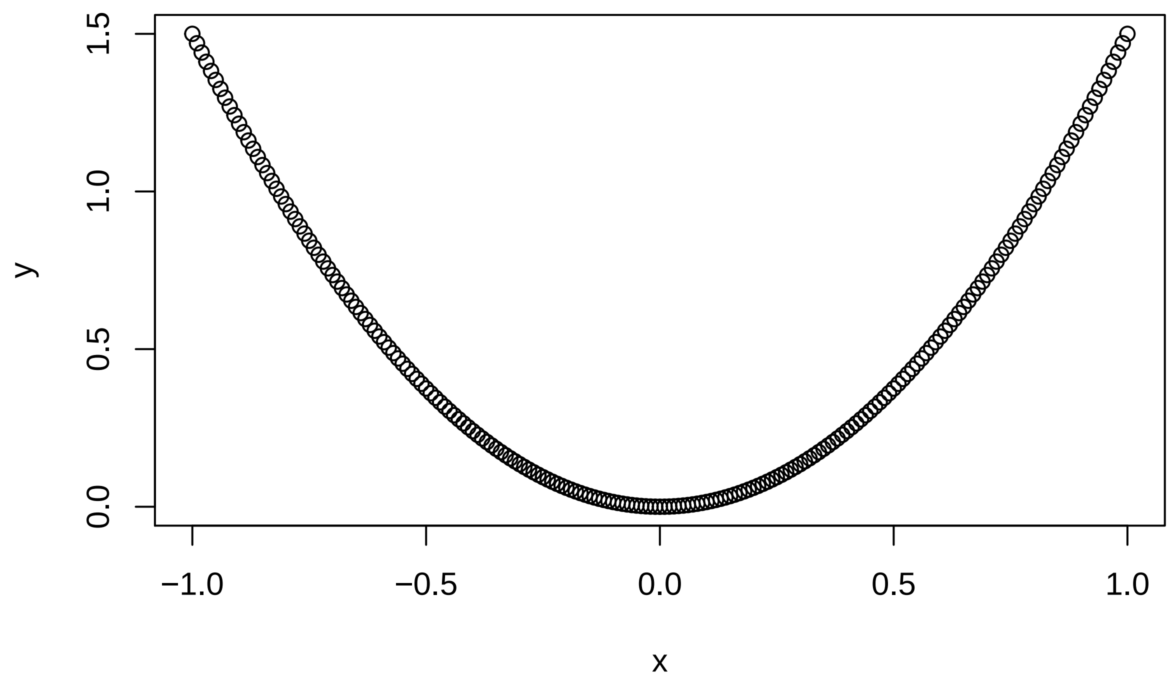
```
rv <- cuberoot(2*x - 1)

hist(rv)
```



Acceptance-Rejection Method

Lets graph our PMF to get an idea of what we should choose for g(x):

```
x <- seq(-1,1,by=0.01)
y <- (3*(x^2))/2

plot(x,y)
```



If would appear the PMF is constantly ≤ 1.5. But, lets analyze the function just to be sure. The derivative of this function is $f'(x) = 3x$. There is a critical point at x=0. The second derivative $f''(x) = 3$, which is positive at x=0, so that point is a minimum. This is the only critical point, so this is a global minimum, and x is increasing as |x| approaches ∞. Since we have a defined range, lets calculate our PMF at the extrema:

$$f(-1) = \frac{3 * (-1)^2}{2} = \frac{3}{2}$$
$$f(1) = \frac{3 * (1)^2)}{2} = \frac{3}{2}$$

Since outside of the interval [-1,1], f(x) = 0, we can state:

$$f(x) \leq 1.5, -\infty < x < \infty$$

If we selected a uniformly distributed random variable ranging from -1 to 1, it's pmf would be:

$$g(x) = 0.5, -1 \leq x \leq 1$$
$$g(x) = otherwise$$

This is because the integral of the entire range of the uniform distribution has to be 1. The uniform distribution with a defined range is a rectangle within that defined range. Since this is ranging from -1 to 1, the height has to satisfy the below integral:

$$\int_{-\infty}^{\infty} g(x) = 1$$

if $g(x) = 0.5$, the above integral will be satisfied.

Given our f(x) and g(x), we can be sure that:

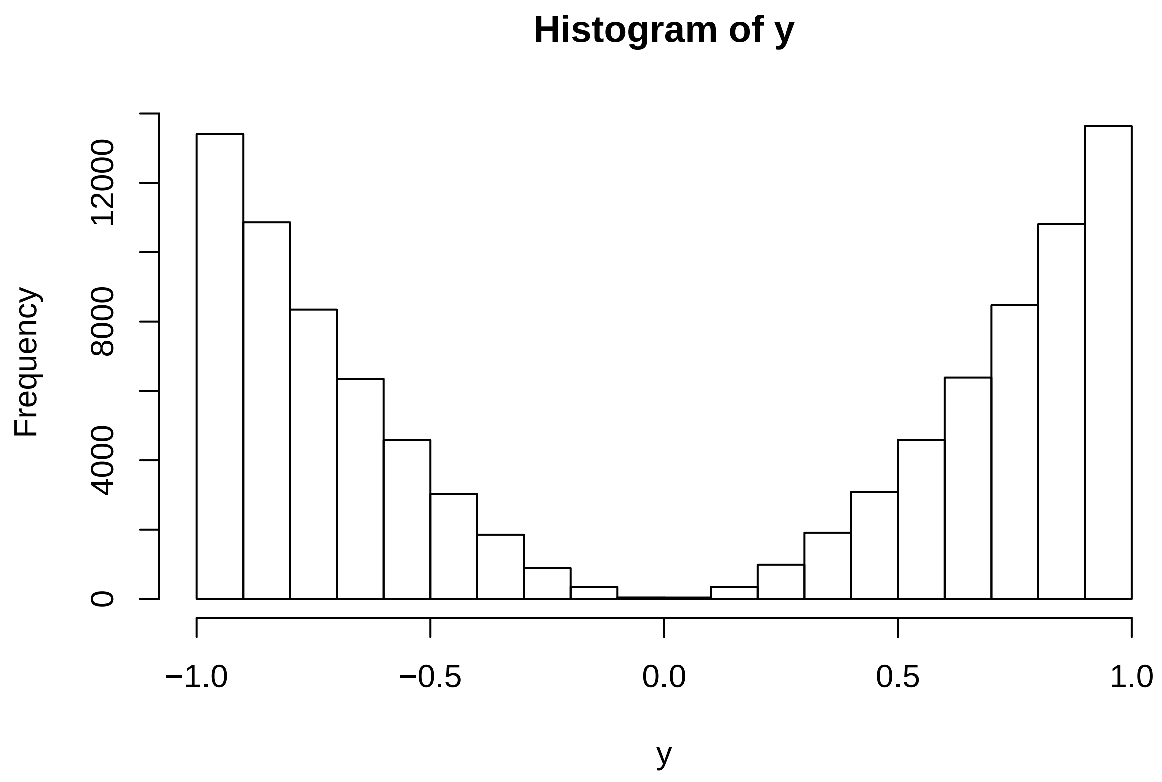
$$\frac{f(x)}{g(x)} \leq \frac{1.5}{0.5} \leq 3$$

So, to sample from f(x), we're going to generate y from g(x), generate a random u from Uniform(0,1), and accept y if $u < \frac{f(y)}{cg(y)}$

For our variate:

$$\frac{f(x)}{cg(x)} = \frac{3x^2}{2} \times \frac{1}{3 * 0.5} = x^2 > u$$

```r
n <- 100000
k <- 0
j <- 0
y <- numeric(n)

while(k < n){
  u <- runif(1)
  j <- j+1
  x <- runif(1,min=-1,max=1)
  if(x^2 > u){
    k <- k+1
    y[k] <- x
  }
}

hist(y)
```



The best way I can think of implementing composition is by separating the function in half vertically at the Y axis.

The PMF is symmetric about the Y-Axis, so I can run two simulations of the positive half of this distribution, and simply make every member of the second simulation negative.

I'll have to multiply my pmf by 2. I'm taking away half of the pmf, yet I still want the area under the curve from 0 to 1 to be 1. To do this, I can simply multiply my function by 2.

So, the variate I'll be generating is:

$$f(x) = 3x^2, 0 \leq x \leq 1$$
$$f(x) = 0, otherwise$$

With my second round of variate generation, I'll make all my numbers negative. Technically, I'll be generating from the positive function, but by making each number negative I'll actually be generating from the negative side of the function, where $-1 \leq x \leq 0$

I'll have a new CDF in this case:

$$F(x) = 0, -\infty < x < 0$$
$$F(x) = \int_0^1 3x^2 0 \leq x \leq 1$$
$$F(x) = 1, 1 < x < \infty$$

So, this means the CDF is:

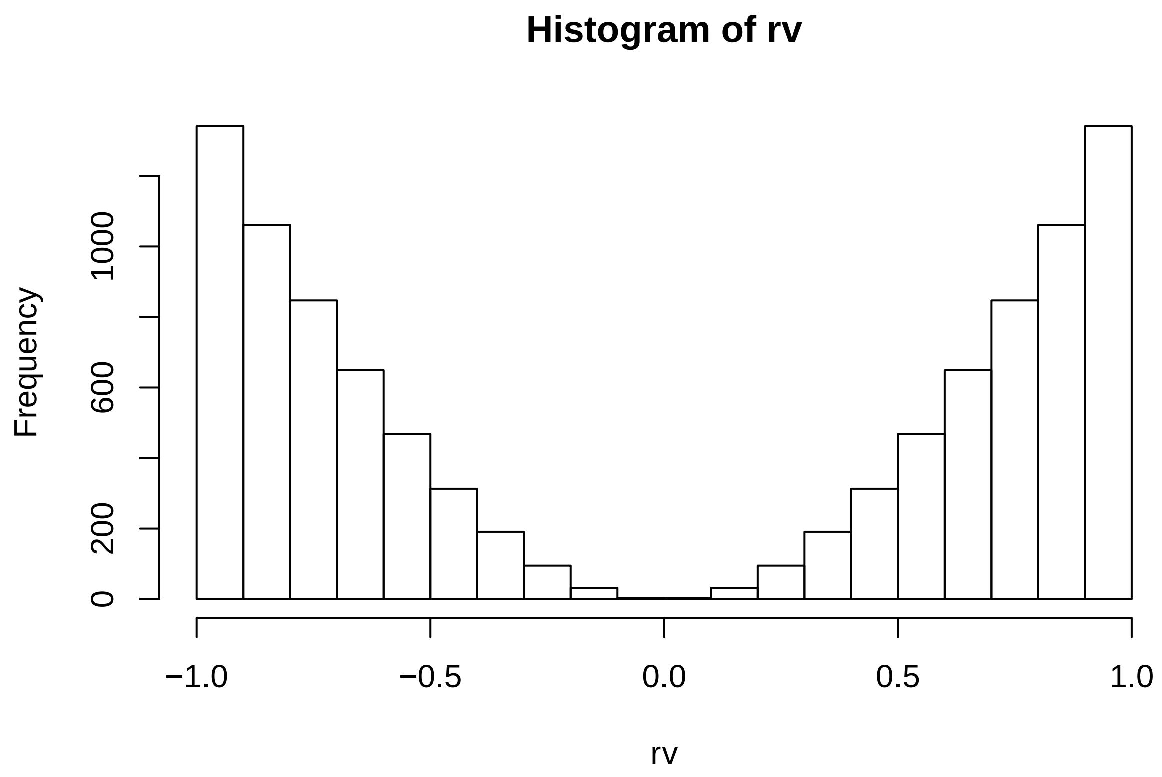
$$F(x) = 0, -\infty < x < 0$$
$$F(x) = x^3, 0 \leq x \leq 1$$
$$F(x) = 1, 1 < x < \infty$$

And to get our inverse $F^{-1}(x)$

$$F^{-1}(x) = (x)^{\frac{1}{3}}, 0 \leq x \leq 1$$
$$F^{-1}(x) = undefined, otherwise$$

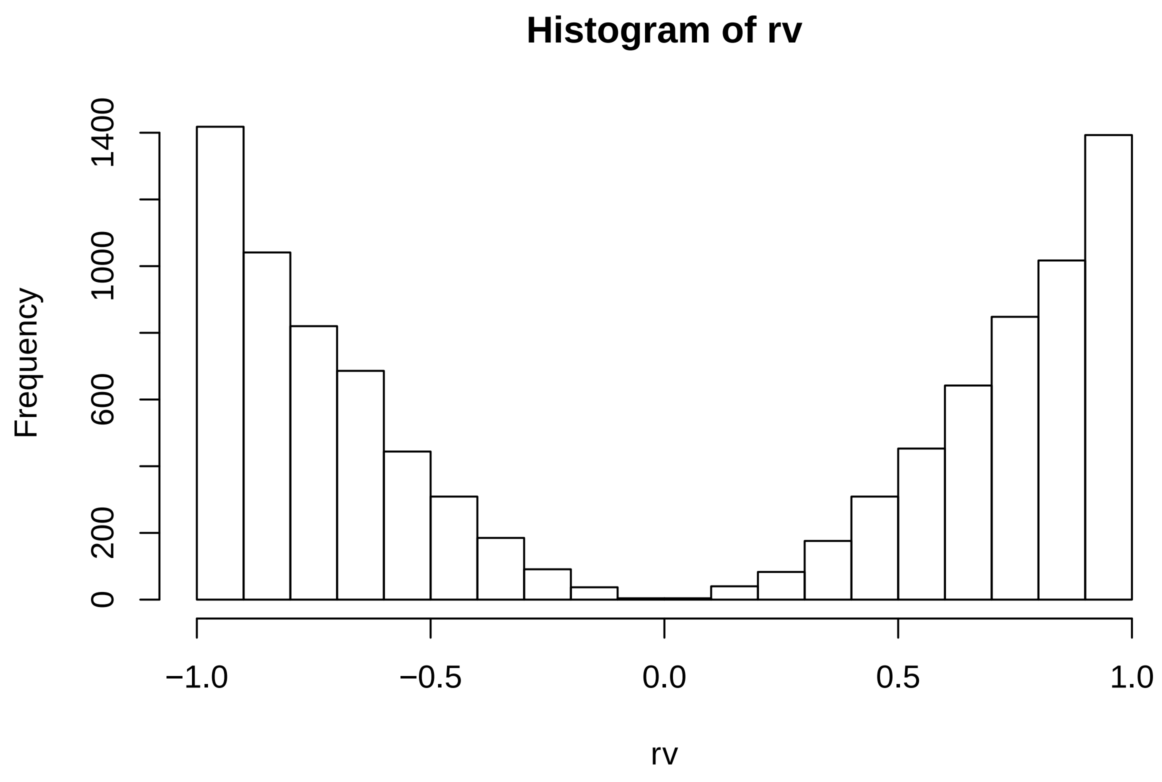
So lets generate this variate:

```r
x <- runif(5000)

rv <- x^(1/3)

xneg <- runif(5000)

rvneg <- -(x^(1/3))

rv <- c(rv,rvneg)

hist(rv)
```



The method I chose has one problem: The negative numbers and positive numbers are grouped together. If I wanted to avoid this issue, I could generate a discrete random variable with equal probabilities of -1 and 1, and multiply that by the full set pulled from my variate ranging from 0 to 1.

```r
neg <- sample(c(1,-1),10000,replace=TRUE)
x <- runif(10000)

rv <- neg*(x^(1/3))

hist(rv)
```



Question 5

Implement, test, and compare different methods to generate from a N(0,1) distribution

Since $\mu = 0$ and $\sigma = 1$, we have the pmf:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

a. The simplest generator is the inverse transform method. Create a function normrandit that:

1. Takes no input variables
2. Generates a random number $U\ U(0,1)$.

3. Returns one output variable: $X = F^{-1}(U)$, where F^{-1} is the inverse normal CDF

We're not able to integrate this over the normal distribution, so I'll use the qnorm function

```
normrandit <- function(){
  U <- runif(1)
  return(qnorm(U))
}
```

Also create a function itstats that:

1. Takes one input variable N
2. Generates N samples from a N(0,1) distribution using normrandit
3. Returns two output variables: the mean and the standard deviation of the samples.

```
itstats <- function(N){
  x <- numeric()
  for(i in 1:N){
    x <- c(x,normrandit())
  }
  return(list(mean=mean(x), sd=sd(x)))
}
```

b. Next up, we want to generate samples using the Box-Muller algorithm. Create a function normrandbm that:

1. Takes no input variables
2. Generates two uniform random variables, U_1 and $U_2 \sim U(0,1)$
3. Returns two output variables: $X = (-2ln(U_1))^{\frac{1}{2}} cos(2\pi U_2)$ and $Y = (-2ln(U_1))^{\frac{1}{2}} sin(2\pi U_2)$

```
normrandbm <- function(){
  U <- runif(2)
  return(((-2*log(U[1]))^(1/2))*cos(2*pi*U[2]))
}
```

As in a), create a function bmstats that can produce N samples using normrandbm and return their mean and the standard deviation.

```
bmstats <- function(N){
  x <- numeric()
  for(i in 1:N){
    x <- c(x,normrandbm())
  }
  return(list(mean=mean(x), sd=sd(x)))
}
```

c. Lastly, we want to generate samples using the accept-reject approach. Create a function normrandar that:

1. Takes no input variable
2. Generates two uniform random numbers: $U_1, U_2\ U(0, 1)$
3. Convert the samples to $Exp(1)$ by calculating $X,Y = -ln(U_i)
4. Accept the sample if $Y \geq \frac{(X-1)^2}{2}$ and reject otherwise
5. If a sample is accepted, randomly choose sign, and return.
6. If sample is rejected, return to 2 and try again.

```r
normrandar <- function(){
  repeat{
    U <- runif(2)
    X <- -log(U[1])
    Y <- -log(U[2])
    if(Y >= ((X-1)^2)/2){
      break
    }
  }
  return(X)
}
```

as in a), create a function arstats that produces N samples using normrandar and returns their means and standard deviations.

```r
arstats <- function(N){
  x <- numeric()
  for(i in 1:N){
    x <- c(x,normrandar())
  }
  return(list(mean=mean(x), sd=sd(x)))
}
```

d. We now compare and evaluate the approaches implemented in parts a, b, and c. Run 10 iterations of itstats, bmstats, and arstats for N = 100, 1000, 10000, and 100000, and calculate the average means and standard deviations produced by each method at each value of N.

In addition, measure the exact CPU time required for each iteration, and calculate the average time required for each method at each value of N.

For each method, plot the average means and standard deviations against the sample size. Which of the three methods appears to be the most accurate? Also, plot the average CPU time vs. N. Which of the three methods appears to take the least time?

```r
library(dplyr)
library(plyr)

results <- expand.grid(1:10,c(100,1000,10000,100000),c("ar","bm","it"))
results$id <- 1:length(results$n)

addmeansdsystime <- function(data){
  row <- ifelse(c(results$method=="ar",results$method=="ar"), as.numeric(arstats(data$n)),
              ifelse(c(results$method=="bm",results$method=="bm"), as.numeric(bmstats(data$n)),
                     as.numeric(itstats(data$n))))
  #row <- as.numeric(arstats(data$n))
  row <- c(row,as.numeric(system.time(arstats(data$n))[3]))
  return(row)
}

testresults <- ddply(results,.variables="id",.fun=addmeansdsystime)
colnames(testresults) <- c("id", "mean", "sd", "system.time")

testresults2 <- merge(results, testresults, by="id")
```

```
# This was my first solution, just kept here in case anyone is interested:

for(n in c(100,1000,10000,100000)){
  means <- data.frame(ar=c(), bm=c(), it=c())
  sds <- data.frame(ar=c(), bm=c(), it=c())
  systimes <- data.frame(ar=c(), bm=c(), it=c())
  for(i in 1:10){
    arlist <- arstats(n)
    bmlist <- bmstats(n)
    itlist <- itstats(n)
    means <- rbind(means,c(arlist$mean, bmlist$mean, itlist$mean))
    sds <- rbind(sds,c(arlist$sd, bmlist$sd, itlist$sd))
    systimes <- rbind(systimes,c(as.numeric(system.time(arstats(n))[3]),
                                 as.numeric(system.time(bmstats(n))[3]),
                                 as.numeric(system.time(itstats(n))[3])))
  }
  colnames(means) <- c("ar","bm", "it")
  colnames(sds) <- c("ar","bm", "it")
  colnames(systimes) <- c("ar","bm", "it")

  newdata <- data.frame(method=c("ar", "bm", "it"), n=rep(n,3),
                        mean=c(mean(means$ar), mean(means$bm), mean(means$it)),
                        sd = c(mean(sds$ar), mean(sds$bm), mean(sds$it)),
                        system.time = c(mean(systimes$ar), mean(systimes$bm), mean(systimes$it)))

  results <- rbind(results, newdata)
}
```

Because this took a long time to run, I saved the data as a csv, and will import it below;

```
setwd("E:/Downloads/Courses/CUNY/SPS/Git/IS 604 Simulation and Modeling techniques/Homework 3")
results <- read.csv("results.csv")

results
```

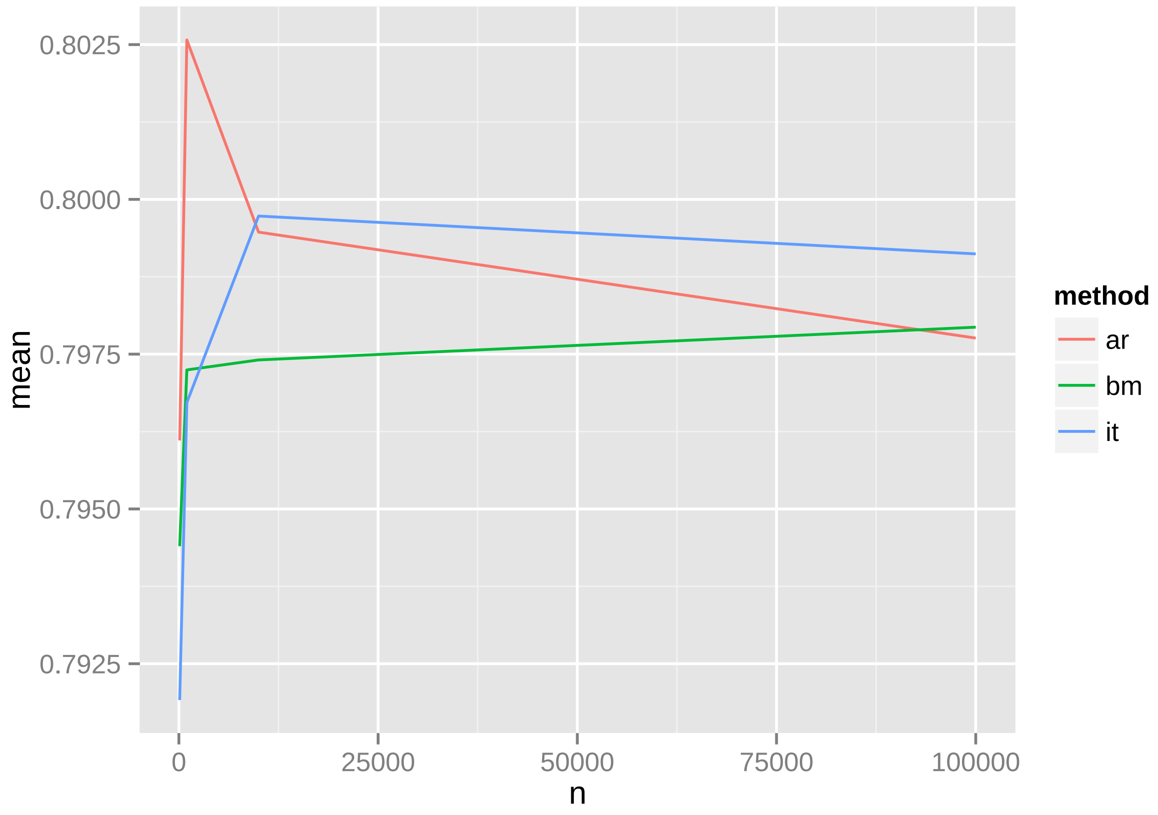
```
##      id trial      n method      mean        sd system.time
## 1     1     1 1e+02     ar 0.8101474 0.5428147        0.00
## 2     2     2 1e+02     ar 0.7665005 0.5472147        0.01
## 3     3     3 1e+02     ar 0.7395028 0.5364303        0.00
## 4     4     4 1e+02     ar 0.7921038 0.6548259        0.00
## 5     5     5 1e+02     ar 0.7664811 0.5736822        0.02
## 6     6     6 1e+02     ar 0.7867099 0.6184116        0.00
## 7     7     7 1e+02     ar 0.8528875 0.5825786        0.00
## 8     8     8 1e+02     ar 0.7864791 0.5460656        0.00
## 9      9     9 1e+02     ar 0.8340104 0.6312963        0.00
## 10    10    10 1e+02     ar 0.8262445 0.6129123        0.02
## 11    11     1 1e+03     ar 0.8099887 0.6164754        0.02
## 12    12     2 1e+03     ar 0.8175645 0.5834161        0.01
## 13    13     3 1e+03     ar 0.8160334 0.6219158        0.02
## 14    14     4 1e+03     ar 0.8147469 0.6097929        0.01
## 15    15     5 1e+03     ar 0.8223350 0.5971562        0.03
## 16    16     6 1e+03     ar 0.7922237 0.6165734        0.01
## 17    17     7 1e+03     ar 0.7697249 0.5714756        0.02
## 18    18     8 1e+03     ar 0.7847561 0.5952407        0.01
## 19    19     9 1e+03     ar 0.8123414 0.6116136        0.02
## 20    20    10 1e+03     ar 0.7860444 0.5941394        0.01
## 21    21     1 1e+04     ar 0.8101722 0.6124302        0.39
## 22    22     2 1e+04     ar 0.7948646 0.6008210        0.42
## 23    23     3 1e+04     ar 0.7985028 0.6073246        0.37
## 24    24     4 1e+04     ar 0.8025815 0.6060227        0.43
## 25    25     5 1e+04     ar 0.7946798 0.6027580        0.42
## 26    26     6 1e+04     ar 0.7945127 0.6000245        0.40
## 27    27     7 1e+04     ar 0.7987087 0.6053431        0.42
## 28    28     8 1e+04     ar 0.7982541 0.6050975        0.41
## 29    29     9 1e+04     ar 0.7947680 0.6032284        0.47
## 30    30    10 1e+04     ar 0.8076675 0.6103520        0.41
## 31    31     1 1e+05     ar 0.7973492 0.6034004       32.55
## 32    32     2 1e+05     ar 0.7995775 0.6019400       32.50
## 33    33     3 1e+05     ar 0.7986322 0.6018179       32.72
## 34    34     4 1e+05     ar 0.7977460 0.6012639       37.66
## 35    35     5 1e+05     ar 0.7972585 0.6020024       44.82
## 36    36     6 1e+05     ar 0.7955979 0.6016144       47.09
## 37    37     7 1e+05     ar 0.7978323 0.6062450       49.66
## 38    38     8 1e+05     ar 0.7989676 0.6036858       36.27
## 39    39     9 1e+05     ar 0.7965911 0.6025881       40.33
## 40    40    10 1e+05     ar 0.7980414 0.6027403       53.93
## 41    41     1 1e+02     bm 0.7976580 0.5689493        0.00
## 42    42     2 1e+02     bm 0.8015198 0.5405553        0.00
## 43    43     3 1e+02     bm 0.8492033 0.6158200        0.00
## 44    44     4 1e+02     bm 0.8145575 0.6130872        0.00
## 45    45     5 1e+02     bm 0.8666473 0.6368761        0.00
## 46    46     6 1e+02     bm 0.7933044 0.6248606        0.00
## 47    47     7 1e+02     bm 0.7631952 0.5944350        0.00
## 48    48     8 1e+02     bm 0.7966113 0.6551043        0.00
## 49    49     9 1e+02     bm 0.6864548 0.5444317        0.00
## 50    50    10 1e+02     bm 0.7748336 0.5338383        0.00
## 51    51     1 1e+03     bm 0.8022778 0.6134325        0.02
## 52    52     2 1e+03     bm 0.8012729 0.5867578        0.02
## 53    53     3 1e+03     bm 0.7899971 0.6174822        0.02
## 54    54     4 1e+03     bm 0.8032765 0.5857161        0.02
## 55    55     5 1e+03     bm 0.7808954 0.5855183        0.02
## 56    56     6 1e+03     bm 0.8170034 0.6069380        0.01
## 57    57     7 1e+03     bm 0.7746790 0.5991872        0.01
## 58    58     8 1e+03     bm 0.8130076 0.6171203        0.02
## 59    59     9 1e+03     bm 0.8143632 0.5962882        0.02
## 60    60    10 1e+03     bm 0.7756840 0.5846471        0.02
## 61    61     1 1e+04     bm 0.8014834 0.6085058        0.39
## 62    62     2 1e+04     bm 0.7962125 0.6033843        0.39
## 63   63      3 1e+04      bm 0.8033867 0.6010796         0.41
## 64   64      4 1e+04      bm 0.7970821 0.6062368         0.41
## 65   65      5 1e+04      bm 0.7987344 0.6047578         0.41
## 66   66      6 1e+04      bm 0.7932260 0.6030382         0.39
## 67   67      7 1e+04      bm 0.8025783 0.6028103         0.39
## 68   68      8 1e+04      bm 0.7902041 0.5961818         0.48
## 69   69      9 1e+04      bm 0.7934685 0.5948387         0.39
## 70   70     10 1e+04      bm 0.7976898 0.6000839         0.41
## 71   71      1 1e+05      bm 0.7968488 0.6047023        39.83
## 72   72      2 1e+05      bm 0.7965235 0.6041829        40.26
## 73   73      3 1e+05      bm 0.7977916 0.6022600        35.24
## 74   74      4 1e+05      bm 0.7965207 0.6027692        35.81
## 75   75      5 1e+05      bm 0.7992494 0.6021668        35.07
## 76   76      6 1e+05      bm 0.7966628 0.6023915        35.40
## 77   77      7 1e+05      bm 0.7993625 0.6038248        36.44
## 78   78      8 1e+05      bm 0.7969523 0.6050318        56.70
## 79   79      9 1e+05      bm 0.7998177 0.6019290        58.95
## 80   80     10 1e+05      bm 0.7996290 0.6044094        63.56
## 81   81      1 1e+02      it 0.8711824 0.6486730         0.00
## 82   82      2 1e+02      it 0.8218441 0.5639842         0.00
## 83   83      3 1e+02      it 0.8201869 0.5719672         0.00
## 84   84      4 1e+02      it 0.7772676 0.5646528         0.00
## 85   85      5 1e+02      it 0.7800879 0.5877659         0.00
## 86   86      6 1e+02      it 0.7841177 0.5431875         0.00
## 87   87      7 1e+02      it 0.7330865 0.5647202         0.02
## 88   88      8 1e+02      it 0.7864689 0.6081861         0.00
## 89   89      9 1e+02      it 0.7075583 0.4820737         0.02
## 90   90     10 1e+02      it 0.8373284 0.6313574         0.00
## 91   91      1 1e+03      it 0.7938379 0.6070435         0.02
## 92   92      2 1e+03      it 0.7744791 0.5959859         0.02
## 93   93      3 1e+03      it 0.7753590 0.5787793         0.03
## 94   94      4 1e+03      it 0.7824202 0.5918242         0.01
## 95   95      5 1e+03      it 0.8300770 0.6391158         0.02
## 96   96      6 1e+03      it 0.7950056 0.6110388         0.01
## 97   97      7 1e+03      it 0.8007007 0.6018891         0.01
## 98   98      8 1e+03      it 0.8044261 0.6017419         0.03
## 99   99      9 1e+03      it 0.8151092 0.6057829         0.03
## 100 100     10 1e+03      it 0.7957676 0.5903588         0.03
## 101 101      1 1e+04      it 0.8054756 0.6045102         0.61
## 102 102      2 1e+04      it 0.8062588 0.6148625         0.56
## 103 103      3 1e+04      it 0.8043857 0.6092787         0.67
## 104 104      4 1e+04      it 0.7963892 0.5992722         0.69
## 105 105      5 1e+04      it 0.7994682 0.6050012         0.56
## 106 106      6 1e+04      it 0.8056063 0.6061955         0.61
## 107 107      7 1e+04      it 0.7907897 0.6014313         0.61
## 108 108      8 1e+04      it 0.7940982 0.5994991         0.59
## 109 109      9 1e+04      it 0.8008010 0.6056033         0.63
## 110 110     10 1e+04      it 0.7940360 0.5929250         0.57
## 111 111      1 1e+05      it 0.8018523 0.6057035       118.20
## 112 112      2 1e+05      it 0.8012054 0.6029594       117.10
## 113 113      3 1e+05      it 0.7969659 0.6018968       106.31
## 114 114      4 1e+05      it 0.7968174 0.6028777        99.09
## 115 115      5 1e+05      it 0.7993093 0.6031032        37.42
## 116 116      6 1e+05      it 0.7969923 0.6000079        40.03
## 117 117     7 1e+05     it 0.8017111 0.6061766       58.80
## 118 118     8 1e+05     it 0.7977928 0.6022164       57.66
## 119 119     9 1e+05     it 0.7986342 0.6011287       43.44
## 120 120    10 1e+05     it 0.7999182 0.6048090       41.30
```

The above results show all trials. I'll aggregate this to get at the means of each of the 10 trials.

```
library(dplyr)
```

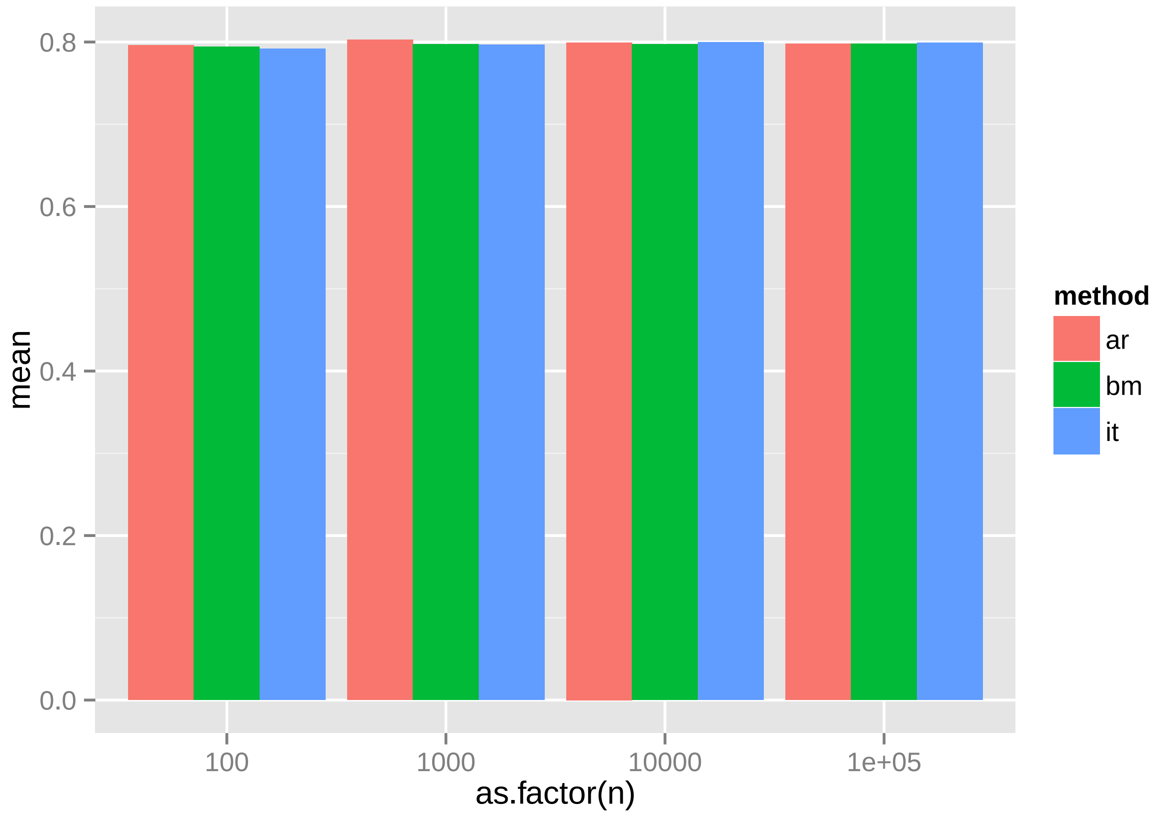
```
##
## Attaching package: 'dplyr'
##
## The following objects are masked from 'package:plyr':
##
##     arrange, count, desc, failwith, id, mutate, rename, summarise,
##     summarize
##
## The following objects are masked from 'package:stats':
##
##     filter, lag
##
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(ggplot2)

resultsagg <- results %>% group_by(n, method) %>% summarize(mean=mean(mean),
                                                             sd=mean(sd),
                                              system.time = mean(system.time))

ggplot(resultsagg,aes(x=n, y=mean, color=method)) + geom_line()
```



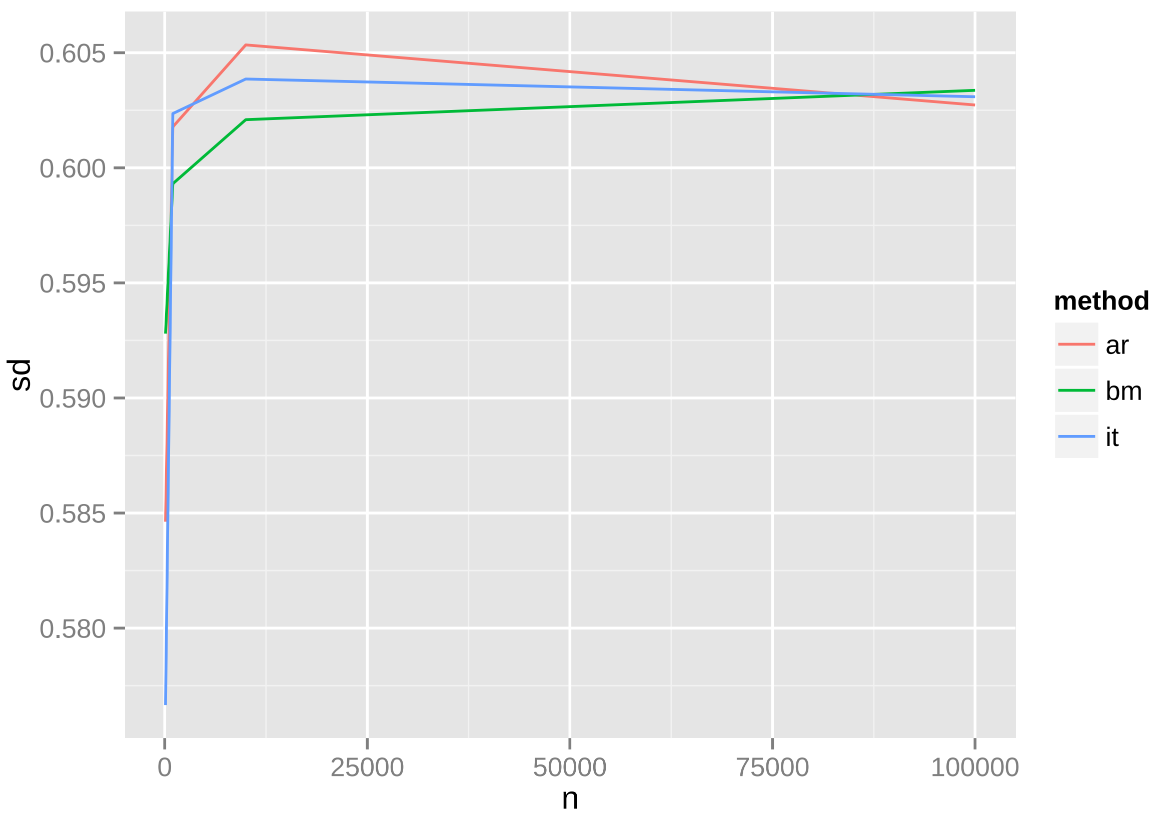
Graphing it out with lines doesn't really provide too much information... Lets try treating n as a factor and graphing it out with bars:

```
ggplot(resultsagg,aes(x=as.factor(n), y=mean, fill=method)) + geom_bar(stat="identity", position="dodge
```

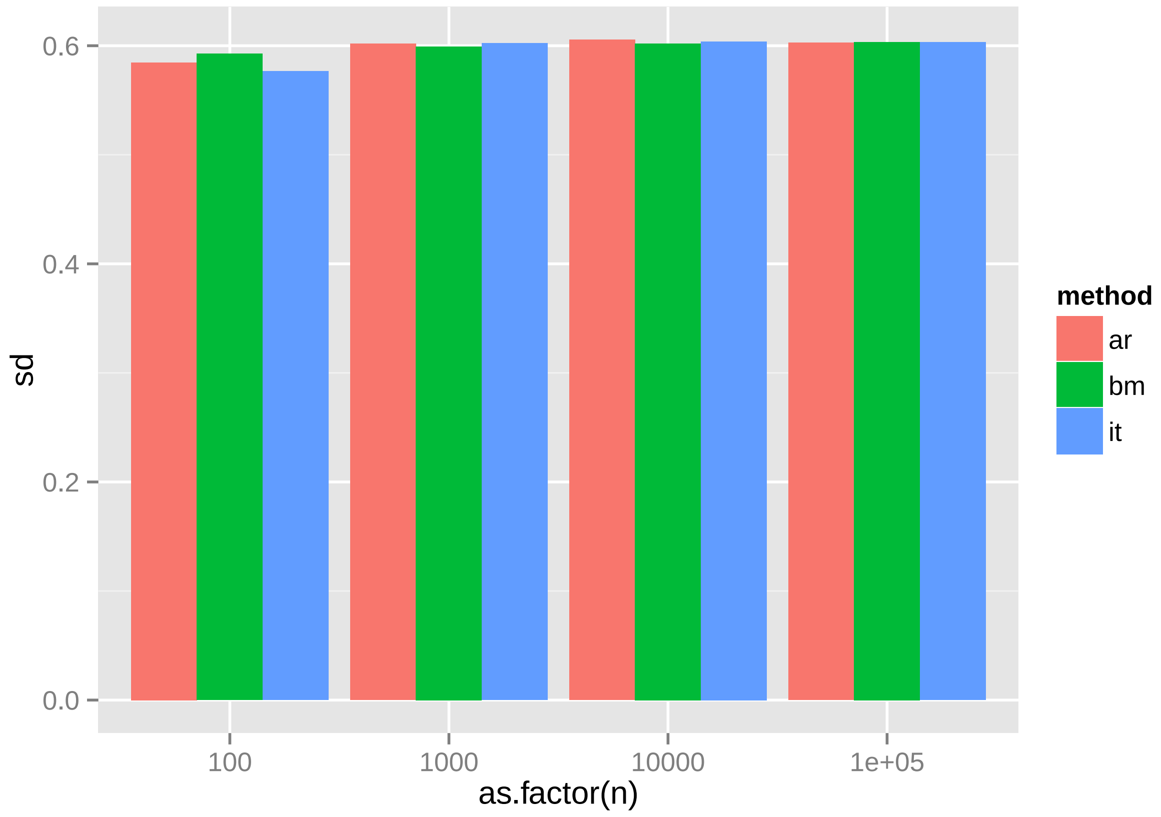


Lets try the same thing with standard deviation:

```
ggplot(resultsagg,aes(x=n, y=sd, color=method)) + geom_line()
```

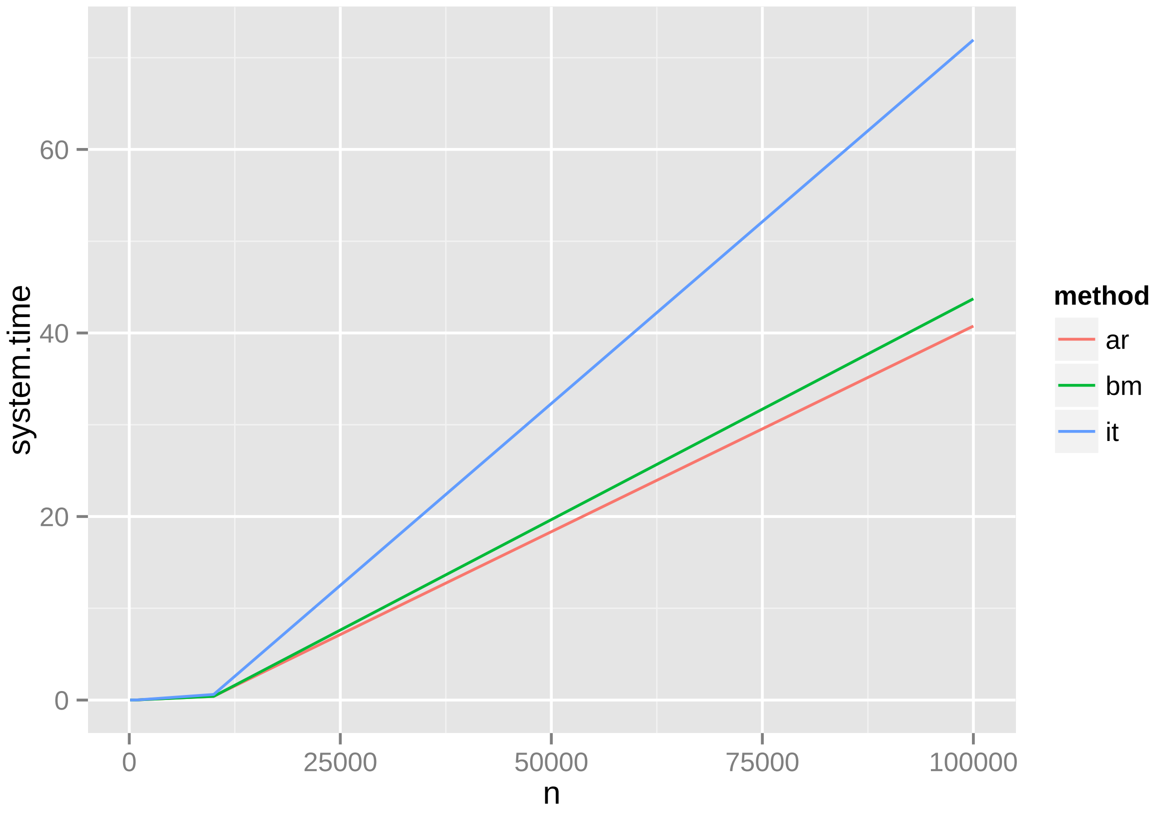


```r
ggplot(resultsagg,aes(x=as.factor(n), y=sd, fill=method)) +
  geom_bar(stat="identity", position="dodge")
```

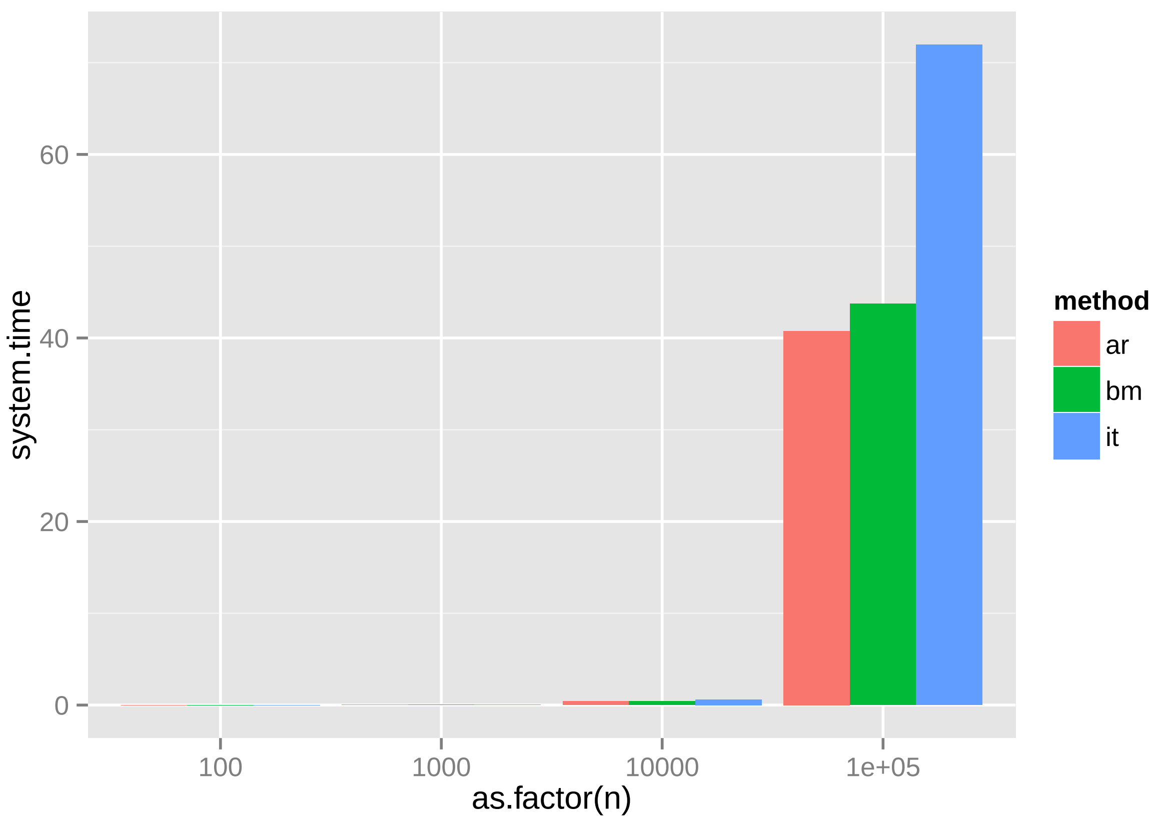


Lets try the same thing with system time:

```
ggplot(resultsagg,aes(x=n, y=system.time, color=method)) + geom_line()
```



```
ggplot(resultsagg,aes(x=as.factor(n), y=system.time, fill=method)) +
  geom_bar(stat="identity", position="dodge")
```



Surprisingly, the Inverse transform method seems to took the longest for me, and Accept-reject was the least.

As expected, increasing N reigned in the mean values. I'm not really sure why the standard deviations increased as N increased…

Question 6

Use Monte-Carlo integration to estimate the value of π.

Generate N pairs of uniform random numbers (x, y) Where x $U(0,1)$ and y $U(0,1)$, and each (x, y) pair represents a point in the unit square. To obtain an estimate of π, count the fraction of points that fall inside the unit quarter circle and multiply by 4. Note that the fraction of points that fall inside the quarter circle should tend to the ratio between the area of the unit quarter circle (i.e.) $\frac{1}{4}\pi$ as compared to the area of the unit square (i.e., 1). We proceed step by step:

a. Create a function insidecircle that takes two inputs between 0 and 1 and returns 1 if these points fall within the unit circle.

```
insidecircle <- function(x,y){
  ifelse(x^2 + y^2 < 1,return(1),return(0))
}
```

b. Create a function estimatepi that takes a single input N, generates N pairs of uniform random numbers, and uses insidecircle to produce an estimate of π, estimatepi should also return the standard error of this estimate, and a 95% confidence interval for the estimate.

```
estimatepi <- function(N){
  x <- runif(N)
  y <- runif(N)
  data <- data.frame(x=x,y=y)
  data$inside <- apply(data,1,function(x) insidecircle(x[1],x[2]))
  mean <- sum(data$inside)/length(data$inside)
  piest <- 4*sum(data$inside)/length(data$inside)
  se <- sd(data$inside)
  pise <- 4*se
  moe <- (1.96*pise)/sqrt(length(data$inside))
  return(list(pi=piest,standard.error=pise,ci95=moe))
}
```

c. Use estimatepi to estimate π for N=1000 to 10000 in increments of 500 and record the estimate, its standard error and the upper and lower bounds of the 95% confidence Interval. How large must N be in order to ensure that your estimate of π is within 0.1 of the true value?

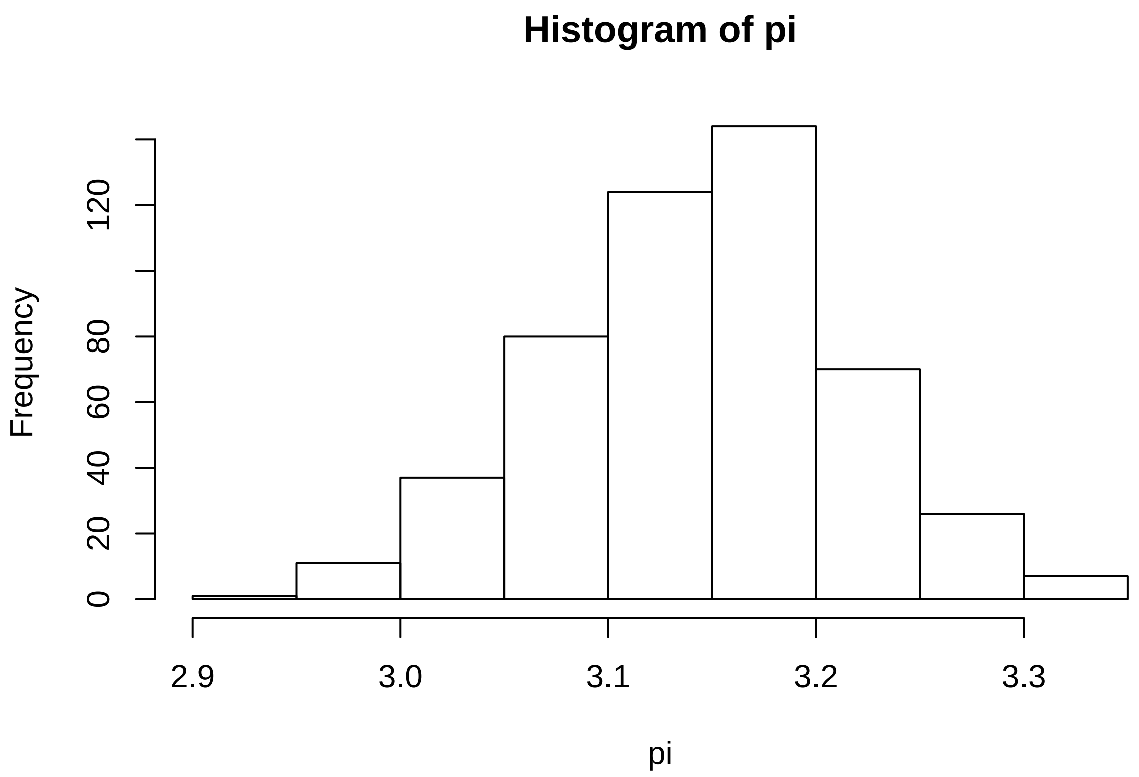
```
citable <- data.frame(n=c(),estimate=c(),se=c(),upper=c(),lower=c(),interval=c())

for(n in seq(1000,10000,by=500)){
  pi <- estimatepi(n)
  citable <- rbind(citable,c(n,pi$pi,pi$standard.error,
                             pi$pi+pi$ci95,pi$pi-pi$ci95,pi$ci95*2))


}

colnames(citable) <- c("number","estimate", "standard.error", "upper", "lower",
                       "interval")

moe4500 <- as.numeric(citable %>% filter(number==4500) %>% select(interval))/2

# for later


citable
```

```
##    number estimate standard.error    upper    lower   interval
## 1    1000 3.060000       1.696844 3.165171 2.954829 0.21034295
## 2    1500 3.112000       1.662919 3.196155 3.027845 0.16831062
## 3    2000 3.096000       1.673375 3.169339 3.022661 0.14667779
## 4    2500 3.172800       1.620367 3.236318 3.109282 0.12703680
## 5    3000 3.153333       1.634231 3.211814 3.094853 0.11696044
## 6    3500 3.148571       1.637544 3.202823 3.094320 0.10850381
## 7    4000 3.157000       1.631569 3.207563 3.106437 0.10112568
## 8    4500 3.119111       1.657770 3.167548 3.070674 0.09687329
## 9    5000 3.100000       1.670496 3.146304 3.053696 0.09260759
## 10   5500 3.141818       1.642176 3.185219 3.098418 0.08680093
## 11   6000 3.110667       1.663394 3.152756 3.068577 0.08417937
## 12   6500 3.142154       1.641919 3.182070 3.102237 0.07983277
## 13   7000 3.162286       1.627719 3.200417 3.124154 0.07626347
## 14   7500 3.142933       1.641360 3.180081 3.105786 0.07429495
## 15   8000 3.138500       1.644432 3.174535 3.102465 0.07207040
## 16   8500 3.115294       1.660254 3.150590 3.079998 0.07059129
## 17   9000 3.148000       1.637802 3.181837 3.114163 0.06767466
## 18   9500 3.121263       1.656218 3.154568 3.087958 0.06661035
## 19  10000 3.142800       1.641425 3.174972 3.110628 0.06434388
```

According to the above table, we must have an N of at least 4500 to be within 0.1 of pi.

d. Using the value of N you determined in part c, run estimatepi 500 times and collect 500 different estimates of pi. Produce a histogram of the estimates and note the shape of this distribution. Calculate the standard deviation of the estimates - does it match up with the standard error you obtained in part c? What percentage of the estimates lie within the 95% CI you obtained in part c?

```
pi <- c()
for(i in 1:500){
  pi <- c(pi,estimatepi(500)$pi)
}

hist(pi)
```



The histogram appears to be normally distributed.

```
sdpi <- sd(pi)

sdpi
```

```
## [1] 0.07121319
```

My standard deviation here, however, seems a lot smaller than the standard error I calculated above. What percentage of the estimates lie within the CI I calculated above?

```r
length(pi[pi>mean(pi)-moe4500 & pi<mean(pi)+moe4500])/length(pi)
```

```
## [1] 0.498
```