IS 604 Assignment 3

David Stern

September 30, 2015

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

```
X <- c(1)
for (i in 1:15){
  X[i+1] <- 11*X[i]%%16
}
X
```

```
##  [1]   1  11 121  99  33  11 121  99  33  11 121  99  33  11 121  99
```

Note, the magnitude of the cycle changes depending on how we express the order of operations. If we put the operations preceding the modulo in parantheses, we get a different sequence. We must be careful to included the parantheses in our syntax so that we properly build Lehmer's linear congruential model.

```
X <- c(1)
for (i in 1:15){
  X[i+1] <- (11*X[i])%%16
}
X
```

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

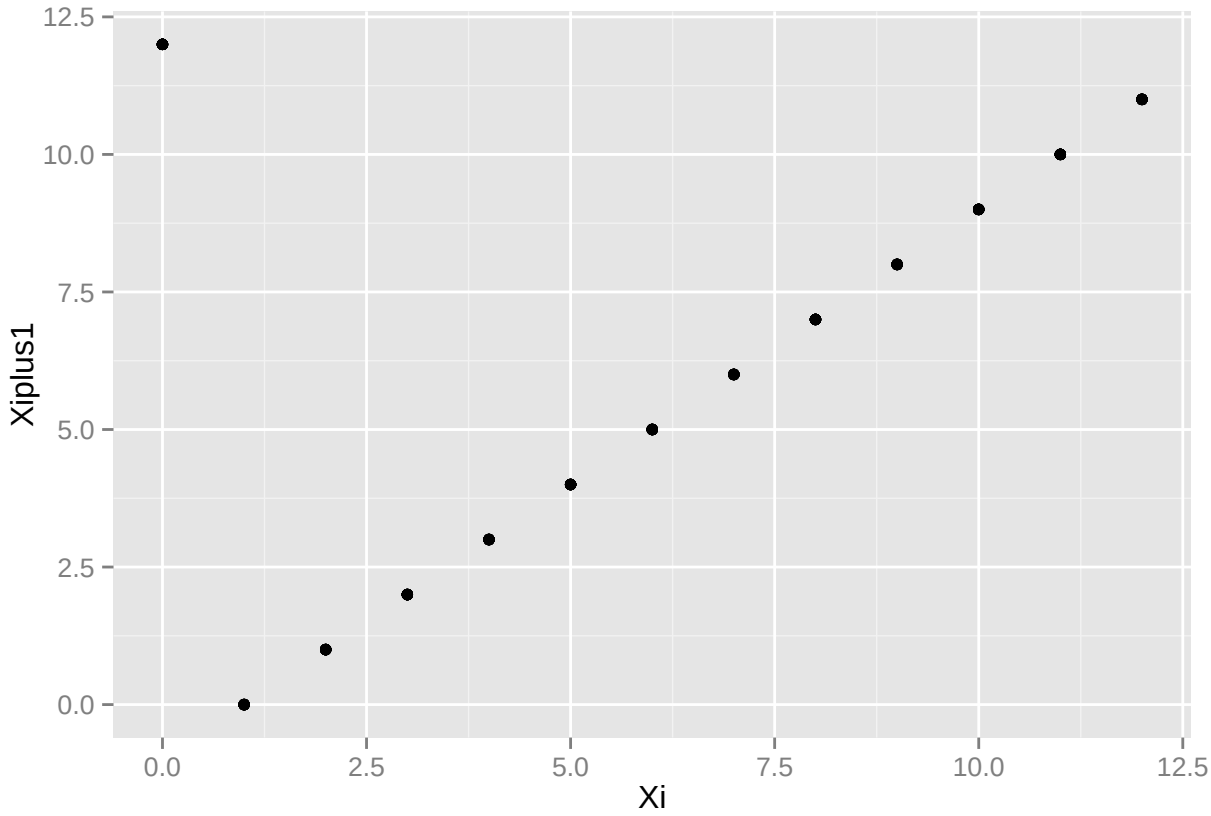
The cycle of random integers we generate is $X = 1, 11, 9, 3....$ Our random numbers are these integers divided by the modulus, $R_i = \frac{X_i}{m} = 0.0625, 0.6875, 0.56250, 0.1875$.

2. Using the LCG provided below: $X_i = (X_{i-1} + 12) mod(13)$, plot the pairs $(U1, U2), (U2, U3)...$ and observe the lattice structure obtained. Discuss what you observed.

```
X <- c(1) # also starting with 1
for (i in 1:200){
  X[i+1] <- (X[i]+12)%%13
}
```

To plot the lattice structure, we will take the vector of X values as X_i and derive a vector of X_{i+1} values by removing the first element of X. To make sure we have a full set of coordinates, we will then have to remove the extra value at the tail end of X.

```
suppressWarnings(suppressMessages(require(ggplot2)))
Xiplus1 <- X[-1]
Xi <- head(X,-1)
lattice <- data.frame(Xi,Xiplus1)
ggplot(lattice) + geom_point(aes(x=Xi,y=Xiplus1))
```



The structure is not a lattice, so much as a line. This happened because there are only 13 unique coordinates among the 200 pairs. We can see this more clearly when we sort the data. Clearly, this is not a very good pesudo-random number generator.

```
lattice[order(Xi),]
```

```
##     Xi Xiplus1
## 2    0      12
## 15   0      12
## 28   0      12
## 41   0      12
## 54   0      12
## 67   0      12
## 80   0      12
## 93   0      12
## 106  0      12
## 119  0      12
## 132  0      12
## 145  0      12
## 158  0      12
## 171  0      12
## 184  0      12
## 197  0      12
## 1    1       0
## 14   1       0
## 27   1       0
## 40   1       0
## 53   1       0
```

```
## 66   1       0
## 79   1       0
## 92   1       0
## 105  1       0
## 118  1       0
## 131  1       0
## 144  1       0
## 157  1       0
## 170  1       0
## 183  1       0
## 196  1       0
## 13   2       1
## 26   2       1
## 39   2       1
## 52   2       1
## 65   2       1
## 78   2       1
## 91   2       1
## 104  2       1
## 117  2       1
## 130  2       1
## 143  2       1
## 156  2       1
## 169  2       1
## 182  2       1
## 195  2       1
## 12   3       2
## 25   3       2
## 38   3       2
## 51   3       2
## 64   3       2
## 77   3       2
## 90   3       2
## 103  3       2
## 116  3       2
## 129  3       2
## 142  3       2
## 155  3       2
## 168  3       2
## 181  3       2
## 194  3       2
## 11   4       3
## 24   4       3
## 37   4       3
## 50   4       3
## 63   4       3
## 76   4       3
## 89   4       3
## 102  4       3
## 115  4       3
## 128  4       3
## 141  4       3
## 154  4       3
## 167  4       3
```

```
## 180  4       3
## 193  4       3
## 10   5       4
## 23   5       4
## 36   5       4
## 49   5       4
## 62   5       4
## 75   5       4
## 88   5       4
## 101  5       4
## 114  5       4
## 127  5       4
## 140  5       4
## 153  5       4
## 166  5       4
## 179  5       4
## 192  5       4
## 9    6       5
## 22   6       5
## 35   6       5
## 48   6       5
## 61   6       5
## 74   6       5
## 87   6       5
## 100  6       5
## 113  6       5
## 126  6       5
## 139  6       5
## 152  6       5
## 165  6       5
## 178  6       5
## 191  6       5
## 8    7       6
## 21   7       6
## 34   7       6
## 47   7       6
## 60   7       6
## 73   7       6
## 86   7       6
## 99   7       6
## 112  7       6
## 125  7       6
## 138  7       6
## 151  7       6
## 164  7       6
## 177  7       6
## 190  7       6
## 7    8       7
## 20   8       7
## 33   8       7
## 46   8       7
## 59   8       7
## 72   8       7
## 85   8       7
```

```
## 98   8       7
## 111  8       7
## 124  8       7
## 137  8       7
## 150  8       7
## 163  8       7
## 176  8       7
## 189  8       7
## 6    9       8
## 19   9       8
## 32   9       8
## 45   9       8
## 58   9       8
## 71   9       8
## 84   9       8
## 97   9       8
## 110  9       8
## 123  9       8
## 136  9       8
## 149  9       8
## 162  9       8
## 175  9       8
## 188  9       8
## 5   10       9
## 18  10       9
## 31  10       9
## 44  10       9
## 57  10       9
## 70  10       9
## 83  10       9
## 96  10       9
## 109 10       9
## 122 10       9
## 135 10       9
## 148 10       9
## 161 10       9
## 174 10       9
## 187 10       9
## 200 10       9
## 4   11      10
## 17  11      10
## 30  11      10
## 43  11      10
## 56  11      10
## 69  11      10
## 82  11      10
## 95  11      10
## 108 11      10
## 121 11      10
## 134 11      10
## 147 11      10
## 160 11      10
## 173 11      10
## 186 11      10
```

```
## 199 11      10
## 3   12      11
## 16  12      11
## 29  12      11
## 42  12      11
## 55  12      11
## 68  12      11
## 81  12      11
## 94  12      11
## 107 12      11
## 120 12      11
## 133 12      11
## 146 12      11
## 159 12      11
## 172 12      11
## 185 12      11
## 198 12      11
```

3. Implement the pseudo-random number generator:

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

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.

```
X <- c(1234567)
for (i in 1:100000){
  X[i+1] <- (16807*X[i])%%((2^31)-1)
}
X[1:10]
```

```
##  [1]    1234567 1422014746  456328559  849987676  681650688 1825340118
##  [7] 1687465831 1569179335 2097898185 1988278849
```

```
R <- X/((2^31)-1)
chisq.test(R)
```

```
## Warning in chisq.test(R): Chi-squared approximation may be incorrect
```

```
##
##  Chi-squared test for given probabilities
##
## data:  R
## X-squared = 16707.99, df = 1e+05, p-value = 1
```

Here I used a double for loop to bin the Random Variables, R_i in 20 equal probability intervals between 0 and 1.

```
intervals <- seq(0,1,by=0.05)
O <- rep(0,20)
for (i in 1:length(R)){
  for (j in 1:20){
  if(intervals[j] < R[i] && R[i] <= intervals[j+1]){
  O[j] <- O[j] + 1
    }
  }
}
O
```

```
##  [1] 5067 5030 5045 5087 4947 4954 4936 4933 4900 4958 5087 4995 5076 5019
## [15] 5003 5067 4980 4919 5058 4940
```

Judging from the values for the number of values in each interval, O_i, the distributions seems to be uniform. Now we can perform the chi square test by taing the sum of the average of the squared differences between the observed number of variables in each class and the expected number of observations. Since we are testing against a uniform distribution we should expect there to be 5,000 observations per class.

$$E_i = \frac{N}{n} = \frac{100,000}{20}$$

$$\chi_0^2 = \Sigma_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

```
sum(((O-5000)^2)/5000)
```

```
## [1] 14.4822
```

Since our calculated value $\chi_O = 14.48$ is much smaller than the value of $\chi_{0.05,9} = 30.1$*, we do not reject the null hypothesis of a uniform distribution is not rejected.

*from table A6 in the Discrete-Event System Simulation textbook

```
suppressWarnings(suppressMessages(require(lawstat)))
runs.test(R)
```

```
##
##  Runs Test - Two sided
##
## data:  R
## Standardized Runs Statistic = -0.3447, p-value = 0.7303
```

Test statistic: $Z = -0.3447$

Significance level: $\alpha = 0.05$

Critical value (upper tail): $Z_{1-\frac{\alpha}{2}} = Z_{0.975} = 1.96$

Critical region: Reject H_0 if $|Z| > 1.96$

Since the absolute value of the test statistic is not greater than 1.96, we will not reject the null hypothesis that the variables were generated in a random manner.

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

$$f(x) = \frac{3x^2}{2} \; for \; -1 \leq x \leq 1, \; otherwise \; 1$$

First we will create an algorithm for the inverse-tranform method:

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

$$F(X) = \frac{x^3}{3} = R$$

$$F^1(x) = (2R)^{\frac{1}{3}}$$

```
inverseTransform <- function(){
 u <- runif(1, min = 0, max = 1)
 x <- (2*u)^(1/3)
 x
}
inverseTransform()
```

```
## [1] 0.6312241
```

Now we will create an algorithm for the acceptance-rejection method. For this algorithm, the function will take one input, the number of samples we want to take, and return the area under the curve by dividing the number of acceptances by the number of samples. With a large number of samples, we should expect the samples to approximate the area under the curve. If we bound bound the graph at m = 1.5, we should expect the acceptance region to approximate 0.33. The area in this region, from $-1 \leq 0 \leq 1$ and $0 \leq y \leq 1.5$ is $2 \times 1.5 = 3$

$$\int_{-1}^{1} \frac{3x^2}{2} = \left[\frac{x^3}{2}\right]_{-1}^{1} = 1$$

$$\frac{Acceptances}{Sample \; Space} = \frac{1}{3} = 0.33$$

```
acceptReject <- function(n){
  counter <- 0
  x <- runif(n,-1,1)
  y <- runif(n,0,1)
  fx <- 1.5*x^2
  for (i in 1:n){if (1.5*y[i] <= fx[i]){counter <- counter + 1}}
  area <- counter/n
  return(area)
                        }
acceptReject(1000)
```

```
## [1] 0.324
```

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

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

1. Takes no input variables.
2. Generates a uniform random number U $U(0,1)$
3. Returns one output variable: $X = F^{-1}(U) where F^{-1}$ is the inverse normal CDF.

For this generator, we will use the r built in *qnorm* to calculate X from the inverse normal CDF.

```
normrandit <- function(){
   U <- runif(1,0,1)
   X <- qnorm(U)
   X
}
normrandit()
```

```
## [1] -0.8572657
```

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){
  samples <- c()
  for(i in 1:n){samples[i] <- normrandit()}
  mean <- mean(samples)
  sd <- sd(samples)
  return(c("mean" = mean,"sd"=sd))
}
itstats(1000)
```

```
##        mean          sd
## -0.01120325  0.99209412
```

Next up, we want to generate samples using the Box-Müller algorithm. Create a function *normrandbm* that: 1. Takes no input variables. 2. Generates two uniform random numbers U_1, U_2 $U(0,1)$ 3. Returns two output variables:

$$X = (-2ln(U_1))^{\frac{1}{2}} cos(2\pi U_2)$$
$$Y = (-2ln(U_1))^{\frac{1}{2}} sin(2\pi U_2)$$

```
normrandbm <- function(){
  U1 <- runif(1,0,1)
  U2 <- runif(1,0,1)
  X <- ((-2*log(U1))^(0.5))*cos(2*pi*U2)
  Y <- ((-2*log(U1))^(0.5))*sin(2*pi*U2)
  variables <- c("X" = X,"Y" = Y)
  variables
}
normrandbm()
```

```
##         X         Y
##  0.692059 -2.000402
```

Create a function $\$bmstats$ that can produce N samples using *normrandbm* and return their mean and the standard deviation

```
bmstats <- function(n){
  sampledf <- data.frame()
  for(i in 1:n){
    newSample <- normrandbm()
    sampledf[i,1] <- as.numeric(newSample[1]) # X is located at first index of the sample
    sampledf[i,2] <- as.numeric(newSample[2]) # Y is located at second index of the sample
               }
    colnames(sampledf) <- c("X","Y")
  meanY <- mean(sampledf$Y)
  sdY <- sd(sampledf$Y)
  meanX <- mean(sampledf$X)
  sdX <- sd(sampledf$X)
  return(c("mean of X" = meanX,"sd of X"=sdX, "mean of Y" = meanY, "sd of Y" = sdY))
}
bmstats(1000)
```

```
##   mean of X     sd of X   mean of Y     sd of Y
## -0.02259607  1.02051784 -0.02794183  1.03889593
```

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 $U1, U2$ $U(0,1)$
3. Converts 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 sample is accepted, randomly choose sign, and return.
6. If sample is rejected, return to 2. and try again.

```
normrandar <- function(){
  U1 <- runif(1,0,1)
  U2 <- runif(1,0,1)
  X <- -log(U1)
  Y <- -log(U2)
  if (Y >= ((X-1)^2)/2){
      if(rnorm(1)<0){X <- -X} # Here I sample from the standard normal distribution for each X and Y
      if(rnorm(1)<0){Y <- -Y} # and pass the sign.
      variables <- c("X" = X,"Y" = Y)
      return(variables)
      }
  else {normrandar()}
}
normrandar()
```

```
##          X          Y
##  0.2435757 -0.9450515
```

Create a function *arstats* that produces N samples using *normrandar* and returns their mean and standard deviation.

```
arstats <- function(n){
  sampledf <- data.frame()
  for(i in 1:n){
    newSample <- normrandar()
    sampledf[i,1] <- as.numeric(newSample[1]) # X is located at first index of the sample
    sampledf[i,2] <- as.numeric(newSample[2]) # Y is located at second index of the sample
                }
    colnames(sampledf) <- c("X","Y")
  meanY <- mean(sampledf$Y)
  sdY <- sd(sampledf$Y)
  meanX <- mean(sampledf$X)
  sdX <- sd(sampledf$X)
  return(c("mean of X" = meanX,"sd of X"=sdX, "mean of Y" = meanY, "sd of Y" = sdY))
}
arstats(1000)
```

```
##    mean of X      sd of X    mean of Y      sd of Y
## -0.015902719  0.994785272 -0.001354466  1.532391940
```

Compare and evaluate the approaches you 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.

Here we will create two functions that take the vector of N values and a stats function as inputs and return a summary matrix of the average mean, standard deviation, and processing time of 10 iterations of the function at each value of N. The first function will evaluate a stats function that returns one variable, X, and the second will evaluate a stats function that returns both X and Y.

To calculate processing time, we will use the *textitsystem.times* function. This function returns an atomic vector of system, user, and elapsed times. We will extract and use the elapsed time for our purposes here.

```
N <- c(100,1000,10000,20000)

tenIterations <- function(method,n){
  allData <- matrix(data=NA, nrow=4, ncol=3) # create summary matrix
  colnames(allData) <- c("mean","sd", "elapsed")
  rownames(allData) <- n
    for(i in 1:length(n)){
      it10 <- replicate(10,method(n[i])) #create temp matrix of mean and sd for 10 iterations of n
      process10 <- c() # create vector of processing times
          for(j in 1:10){process10[j] <- system.time(method(n[i]))["elapsed"]}
      it10 <- rbind(it10,process10) # add vector to matrix
      allData[i,] <- apply(it10,1,mean) #calc. average mean, sd, and proc. time for 10 it, row-wise, &
      }
  allData
}

tenIterations2var <- function(method,n){
  allData <- matrix(data=NA, nrow=4, ncol=5)
  colnames(allData) <- c("meanX","sdX","meanY","sdY", "elapsed")
```

```
  rownames(allData) <- n
    for(i in 1:length(n)){
      it10 <- replicate(10,method(n[i]))
      process10 <- c()
          for(j in 1:10){process10[j] <- system.time(method(n[i]))["elapsed"]}
      it10 <- rbind(it10,process10)
      allData[i,] <- apply(it10,1,mean)
      }
  allData
}

ITdata <- tenIterations(itstats,N)
BMdata <- tenIterations2var(bmstats,N)
ARdata <- tenIterations2var(arstats,N)
ITdata
```

```
##               mean        sd elapsed
## 100   -0.055396950 1.0022773  0.0013
## 1000  -0.001301732 0.9975581  0.0086
## 10000  0.002367770 0.9923002  0.2466
## 20000 -0.001397228 1.0005269  1.0796
```

```
BMdata
```

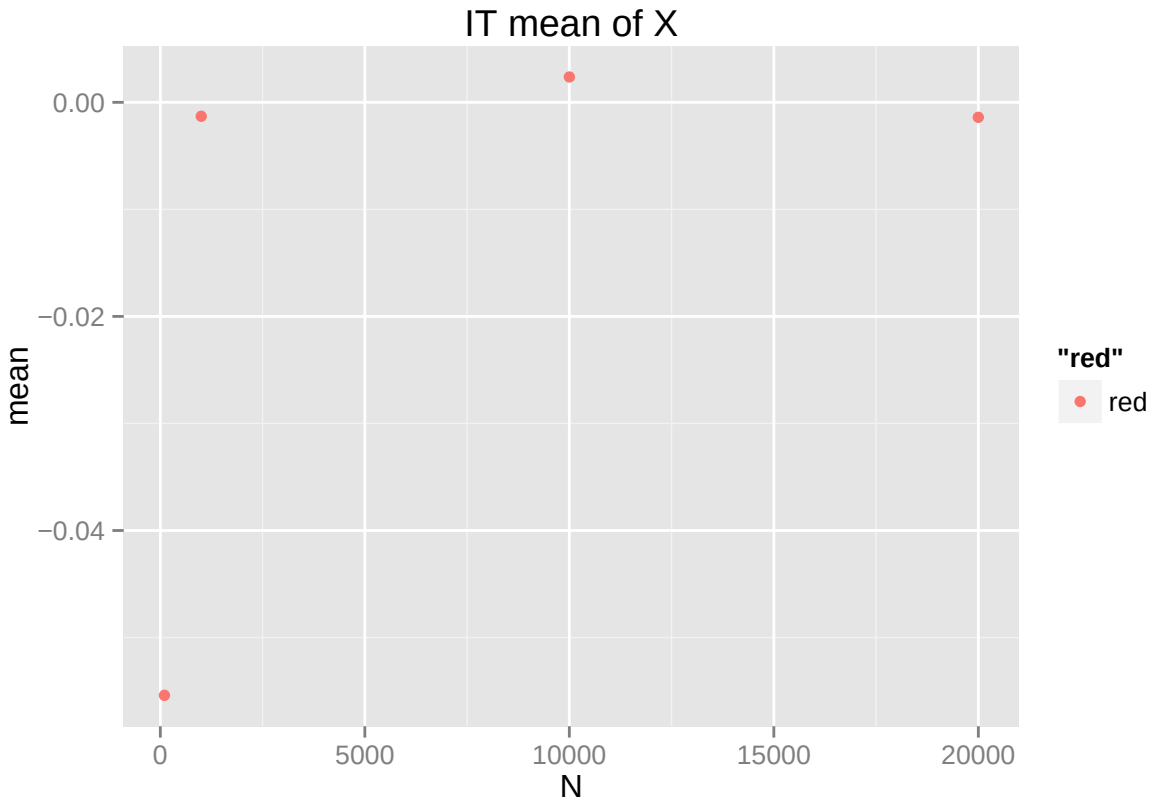
```
##              meanX       sdX        meanY      sdY elapsed
## 100   -0.029436109 0.9747870 -0.006641215 1.055673  0.0131
## 1000   0.029430530 1.0061443 -0.001543197 1.004772  0.1616
## 10000 -0.001617462 1.0026339  0.006137236 1.003003  4.3877
## 20000  0.001487181 0.9998579  0.002211464 0.999627 21.8314
```

```
ARdata
```

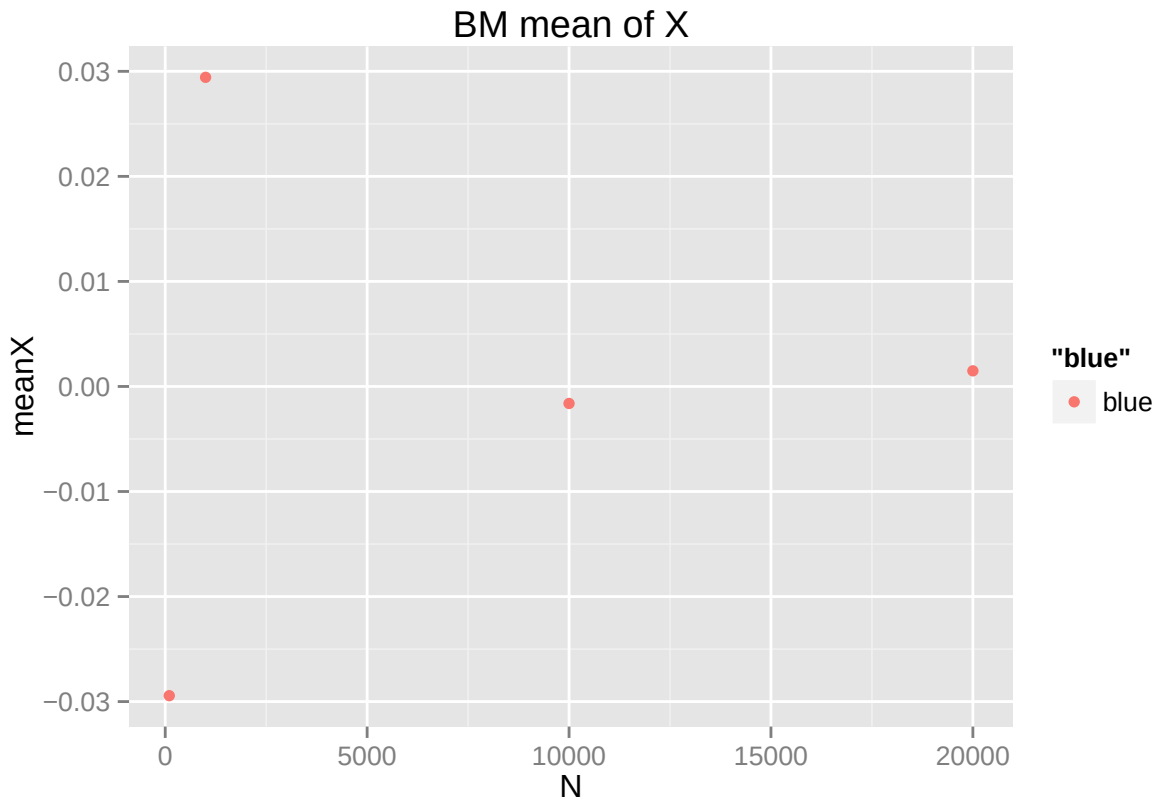
```
##              meanX       sdX        meanY      sdY elapsed
## 100   -0.014251719 1.0189860  0.086256772 1.615600  0.0189
## 1000  -0.005344463 0.9935527 -0.007526696 1.572256  0.1872
## 10000  0.002839633 0.9986008 -0.005810963 1.578583  4.8917
## 20000 -0.001582040 0.9998819 -0.001007865 1.586964 21.6085
```

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?

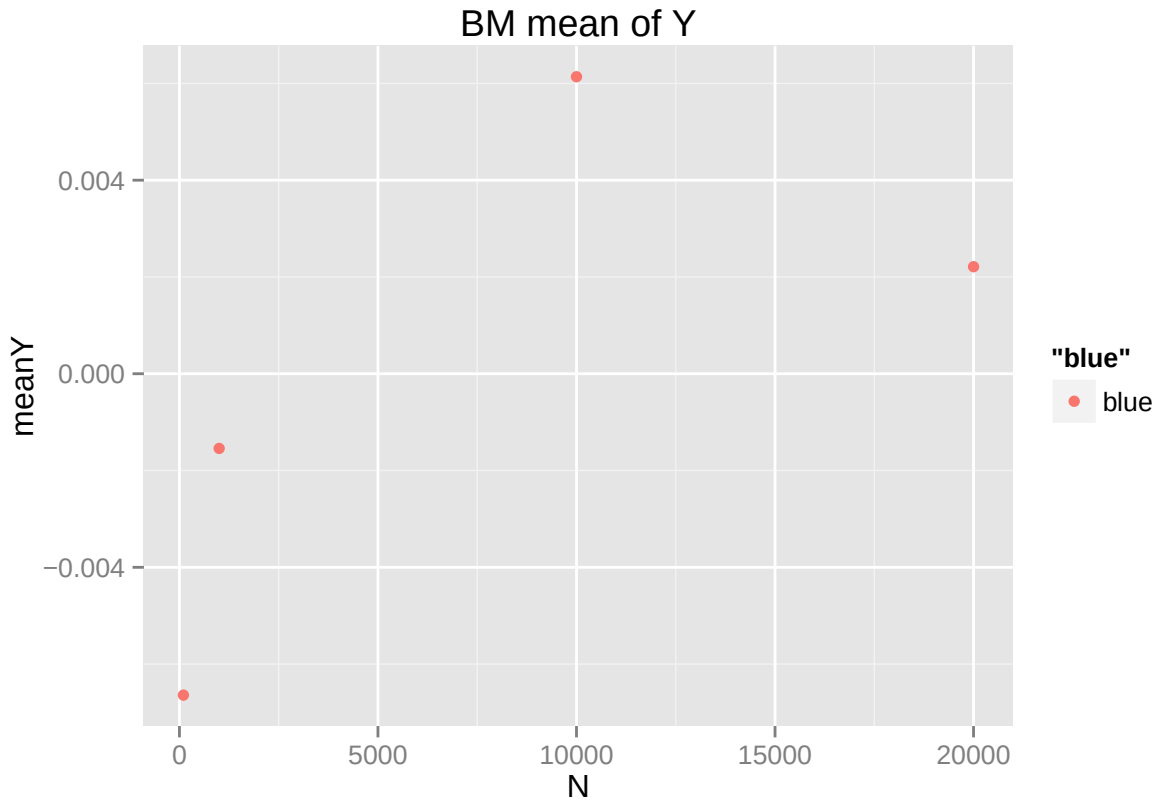
```
suppressWarnings(suppressMessages(require(gridExtra)))
ITdf <- as.data.frame(ITdata)
BMdf <- as.data.frame(BMdata)
ARdf <- as.data.frame(ARdata)
ggplot(ITdf) + geom_point(aes(x=N,y=mean,colour="red")) + ggtitle("IT mean of X")
```



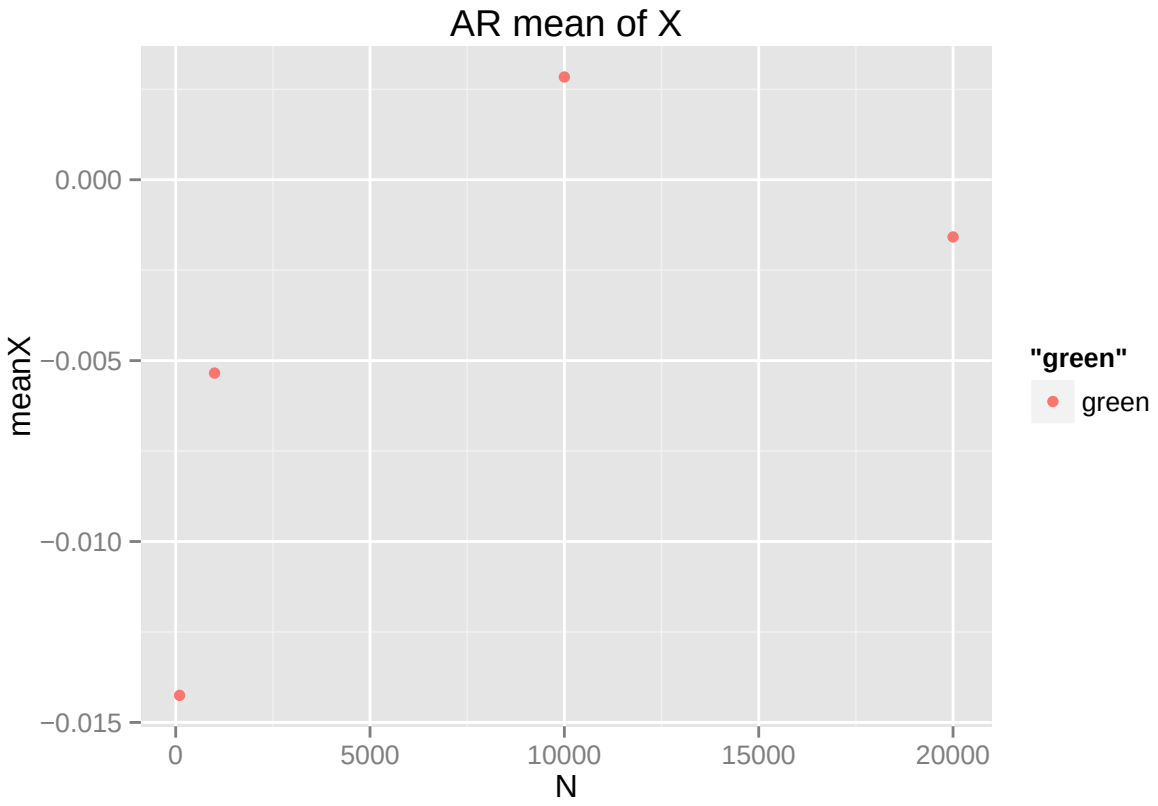

```
ggplot(BMdf) + geom_point(aes(x=N,y=meanX,colour="blue")) + ggtitle("BM mean of X")
```



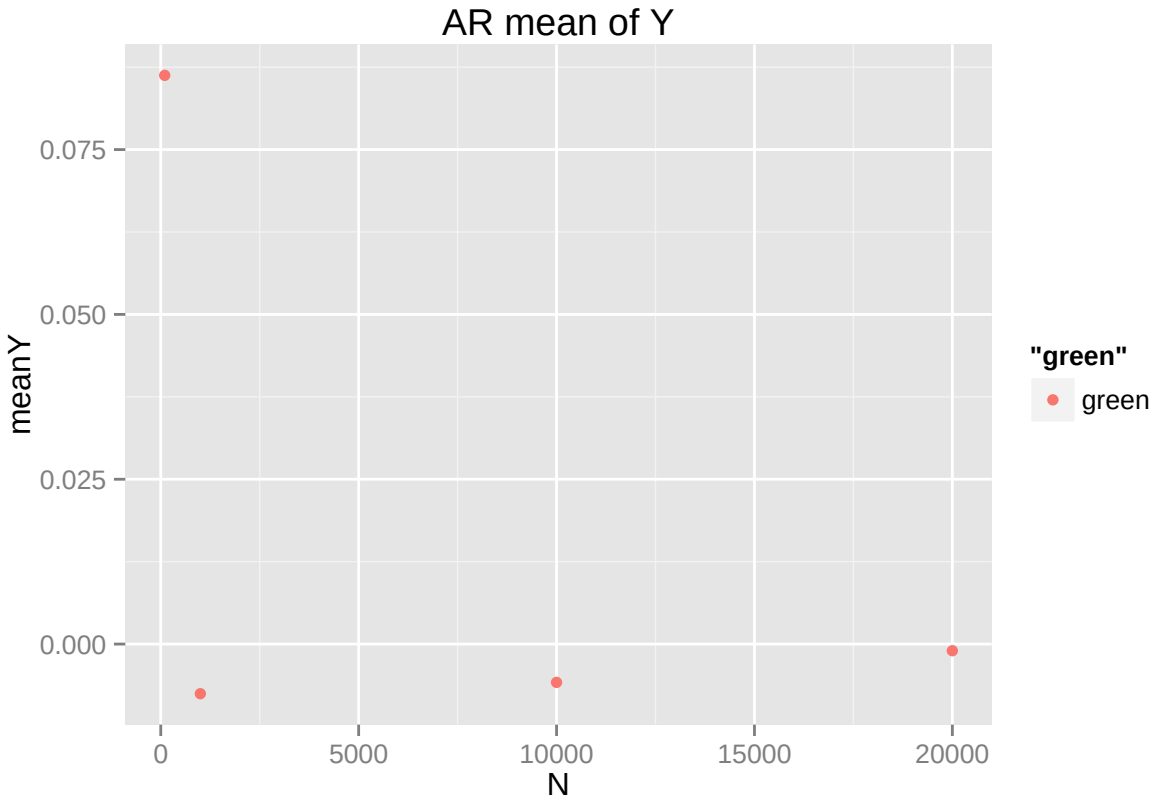
```
ggplot(BMdf) + geom_point(aes(x=N,y=meanY,colour="blue")) + ggtitle("BM mean of Y")
```



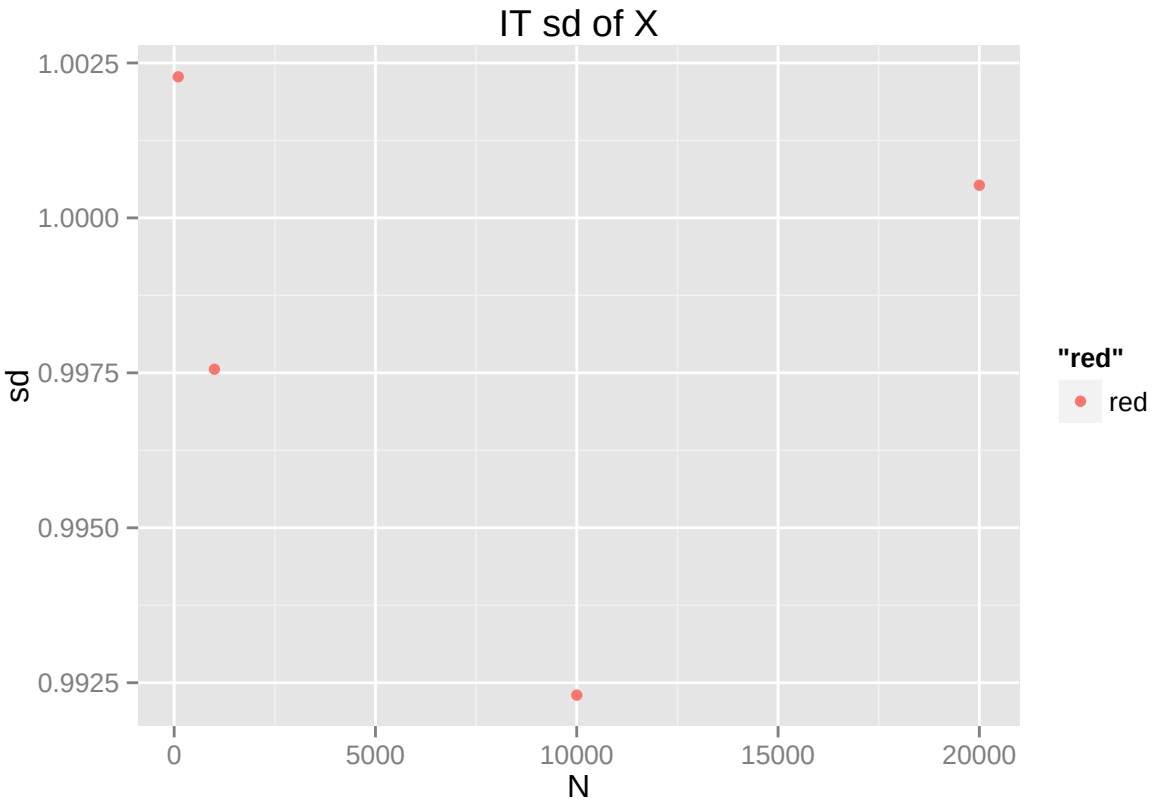

```
ggplot(ARdf) + geom_point(aes(x=N,y=meanX,colour="green")) + ggtitle("AR mean of X")
```



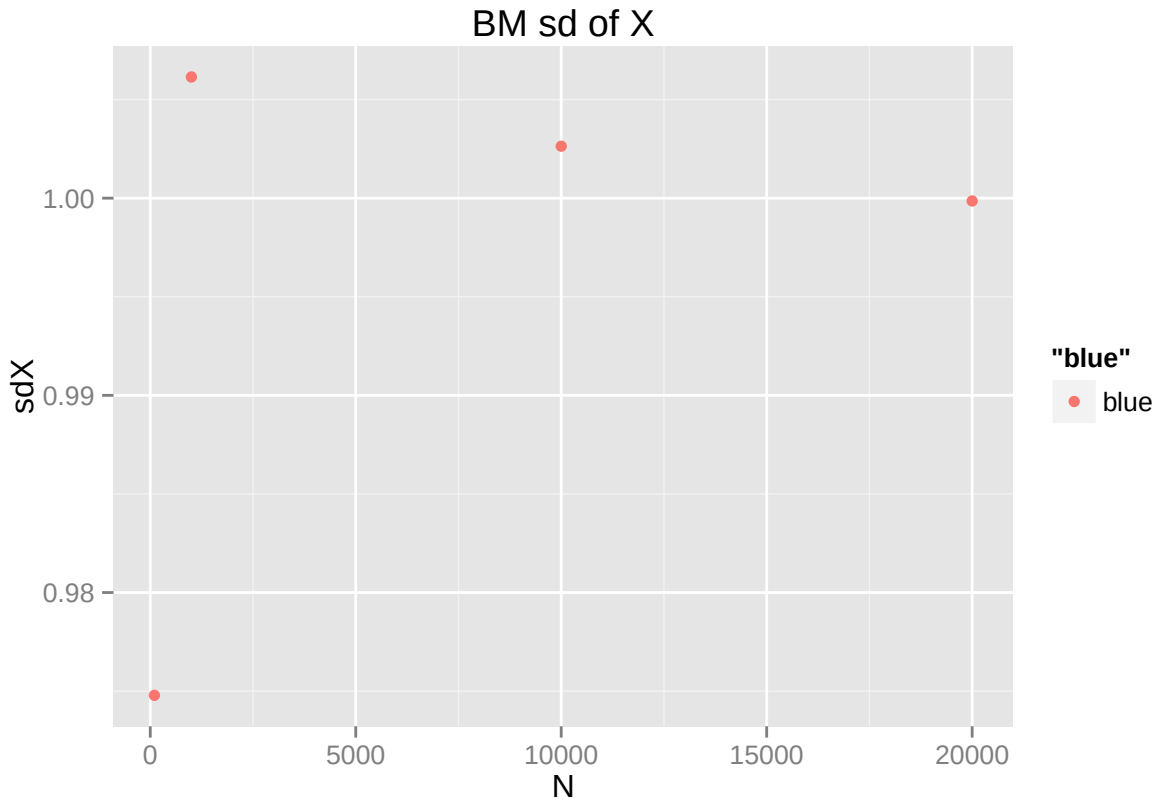

```
ggplot(ARdf) + geom_point(aes(x=N,y=meanY,colour="green")) + ggtitle("AR mean of Y")
```



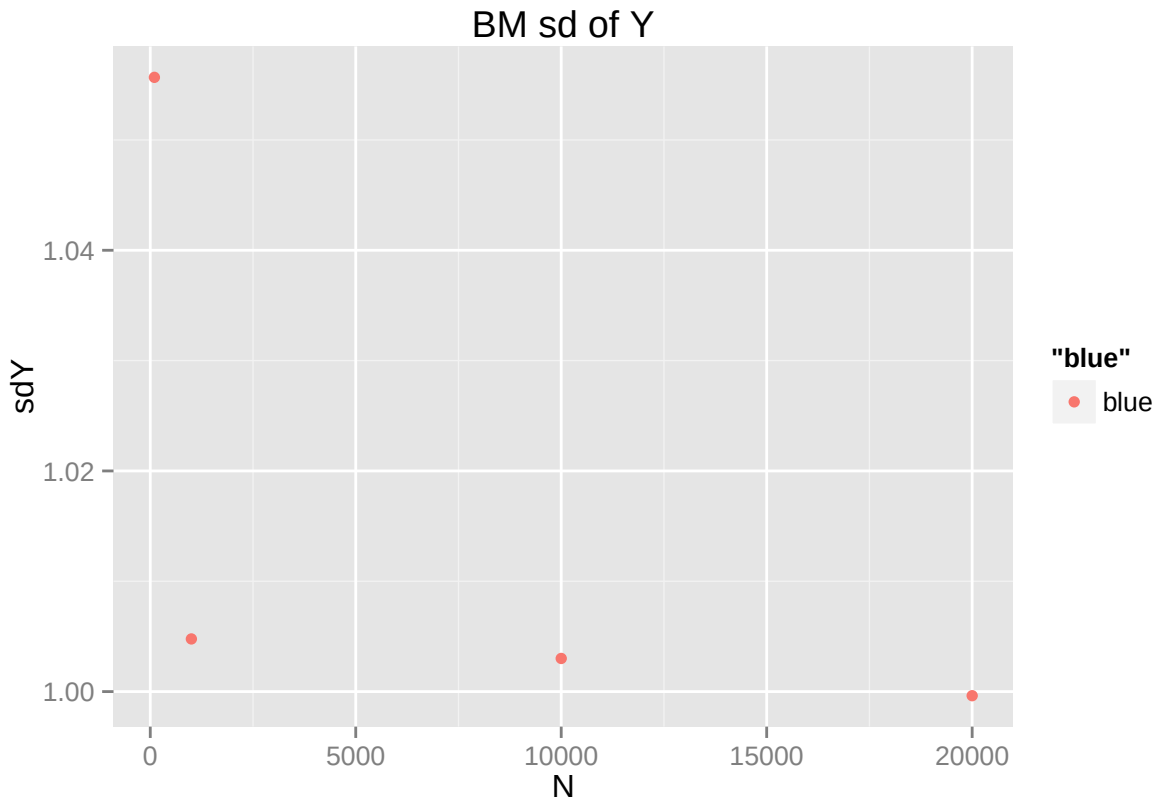
```
ggplot(ITdf) + geom_point(aes(x=N,y=sd,colour="red")) + ggtitle("IT sd of X")
```



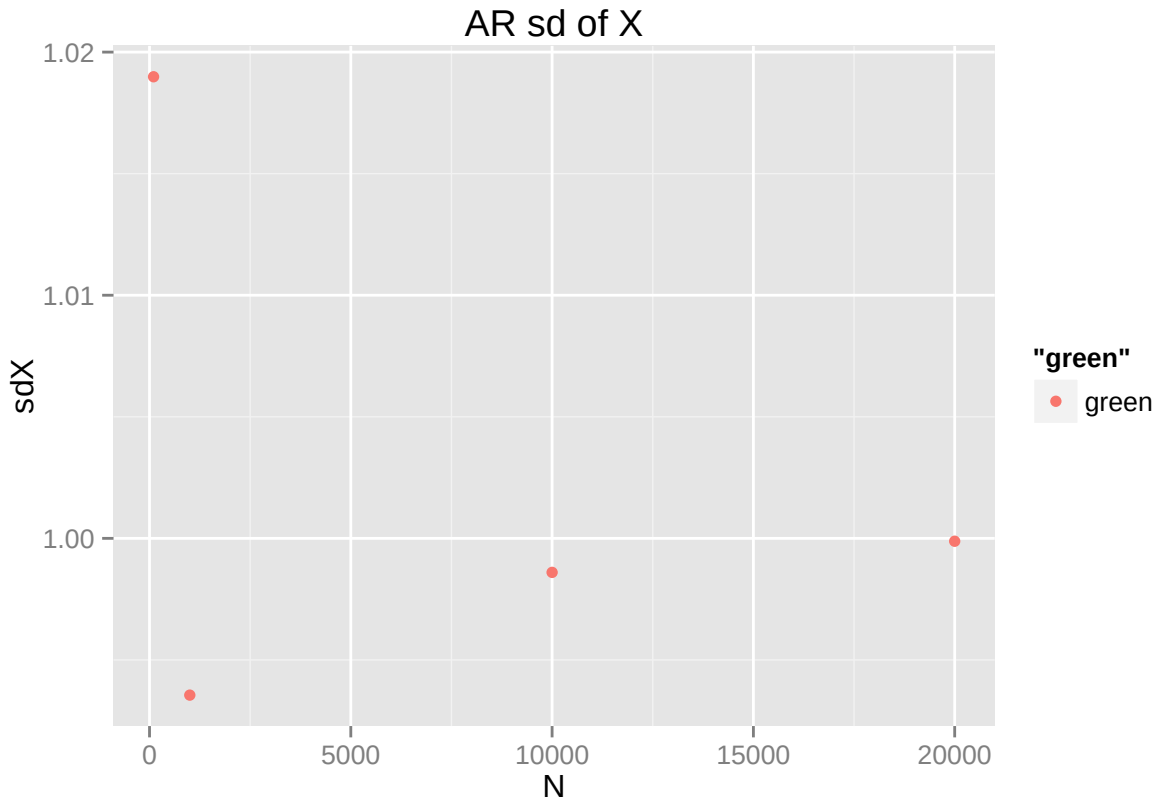

```
ggplot(BMdf) + geom_point(aes(x=N,y=sdX,colour="blue")) + ggtitle("BM sd of X")
```



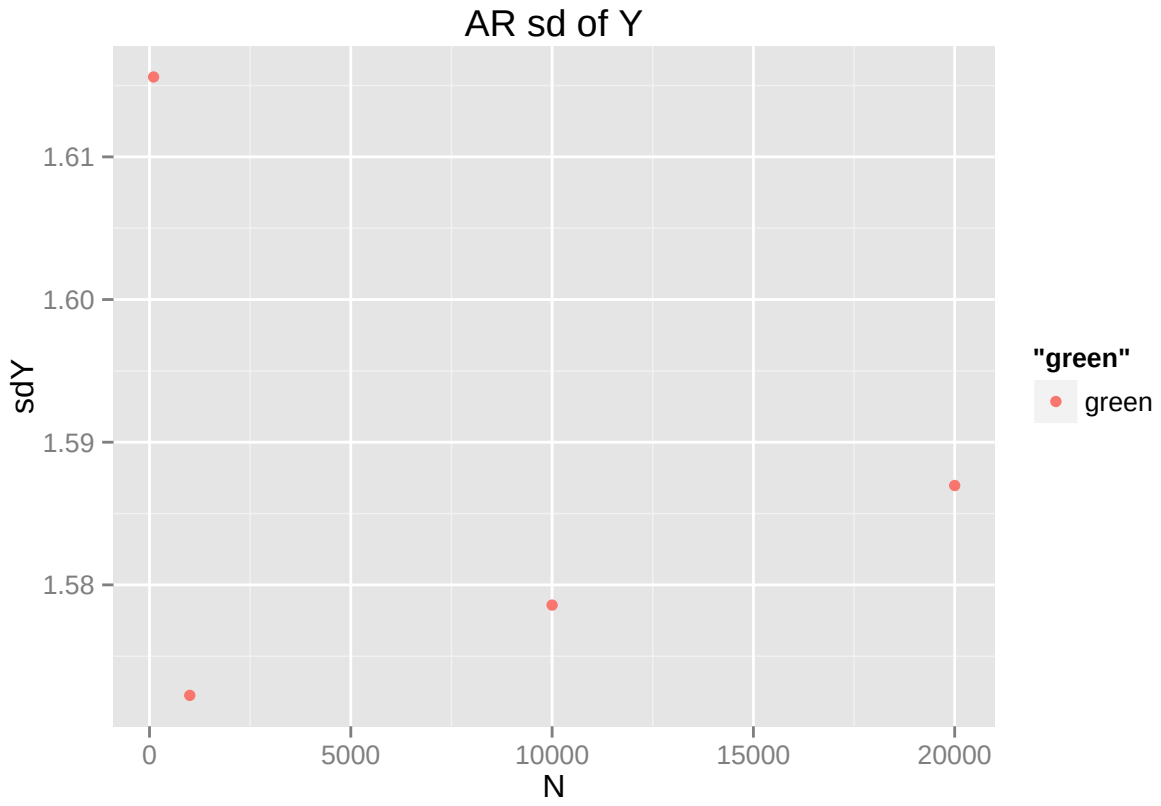

```
ggplot(BMdf) + geom_point(aes(x=N,y=sdY,colour="blue")) + ggtitle("BM sd of Y")
```



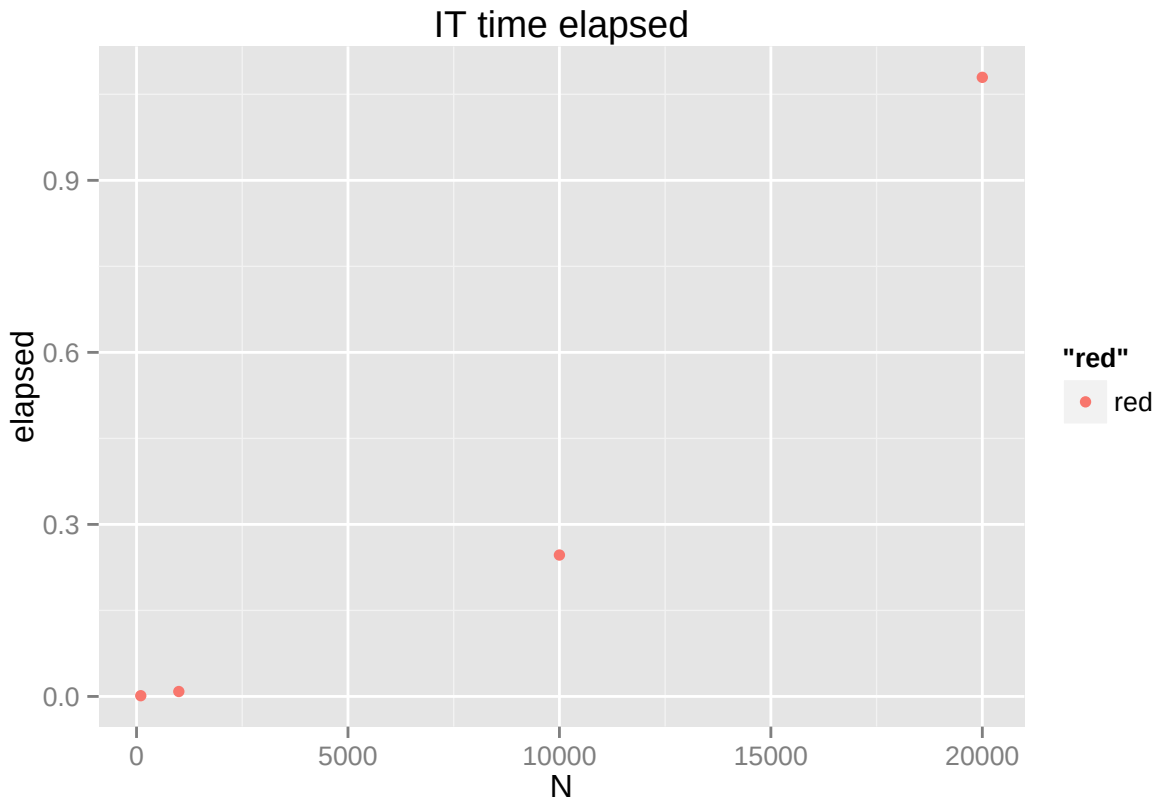
```
ggplot(ARdf) + geom_point(aes(x=N,y=sdX,colour="green")) + ggtitle("AR sd of X")
```



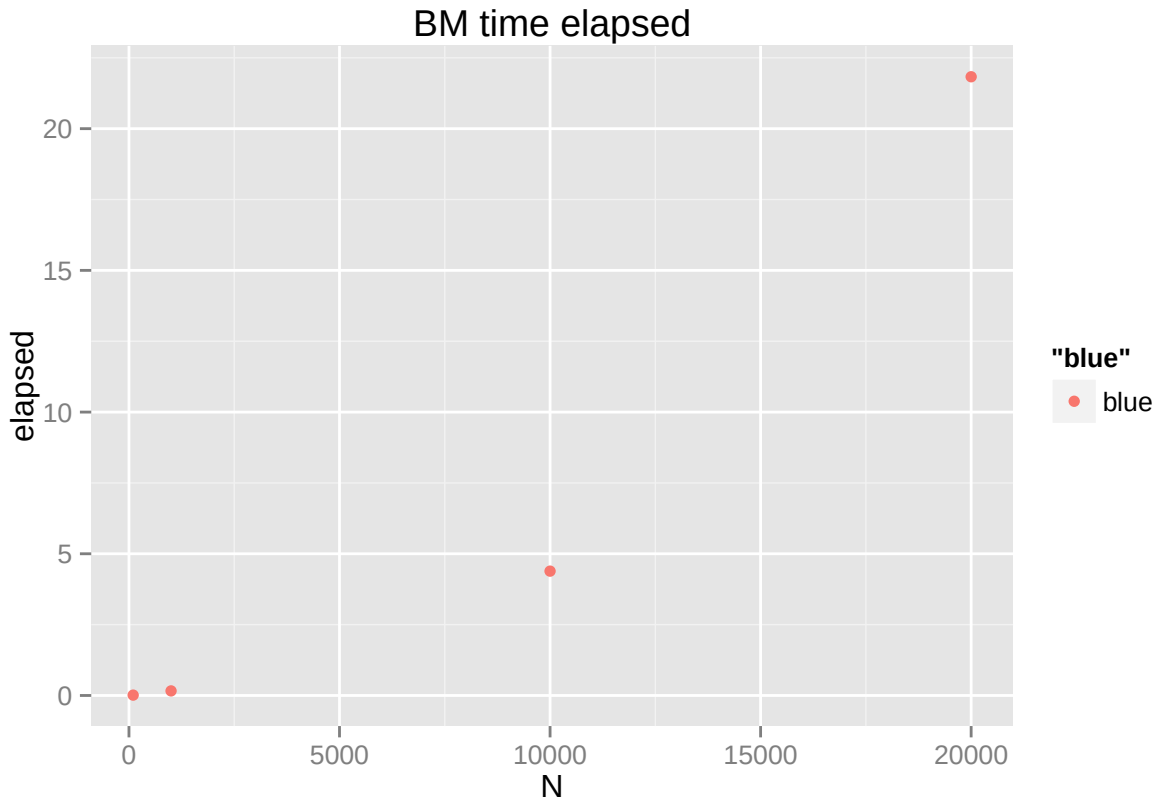

```
ggplot(ARdf) + geom_point(aes(x=N,y=sdY,colour="green")) + ggtitle("AR sd of Y")
```



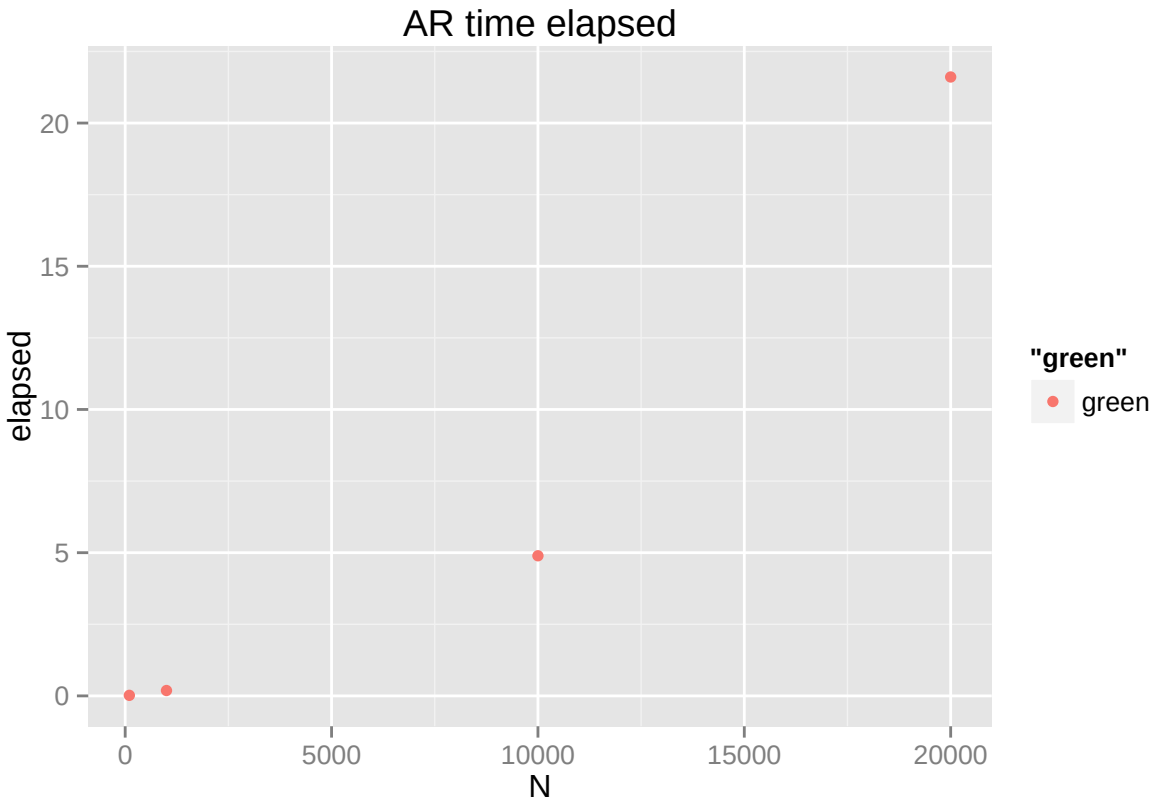

```
ggplot(ITdf) + geom_point(aes(x=N,y=elapsed,colour="red")) + ggtitle("IT time elapsed")
```



```
ggplot(BMdf) + geom_point(aes(x=N,y=elapsed,colour="blue")) + ggtitle("BM time elapsed")
```



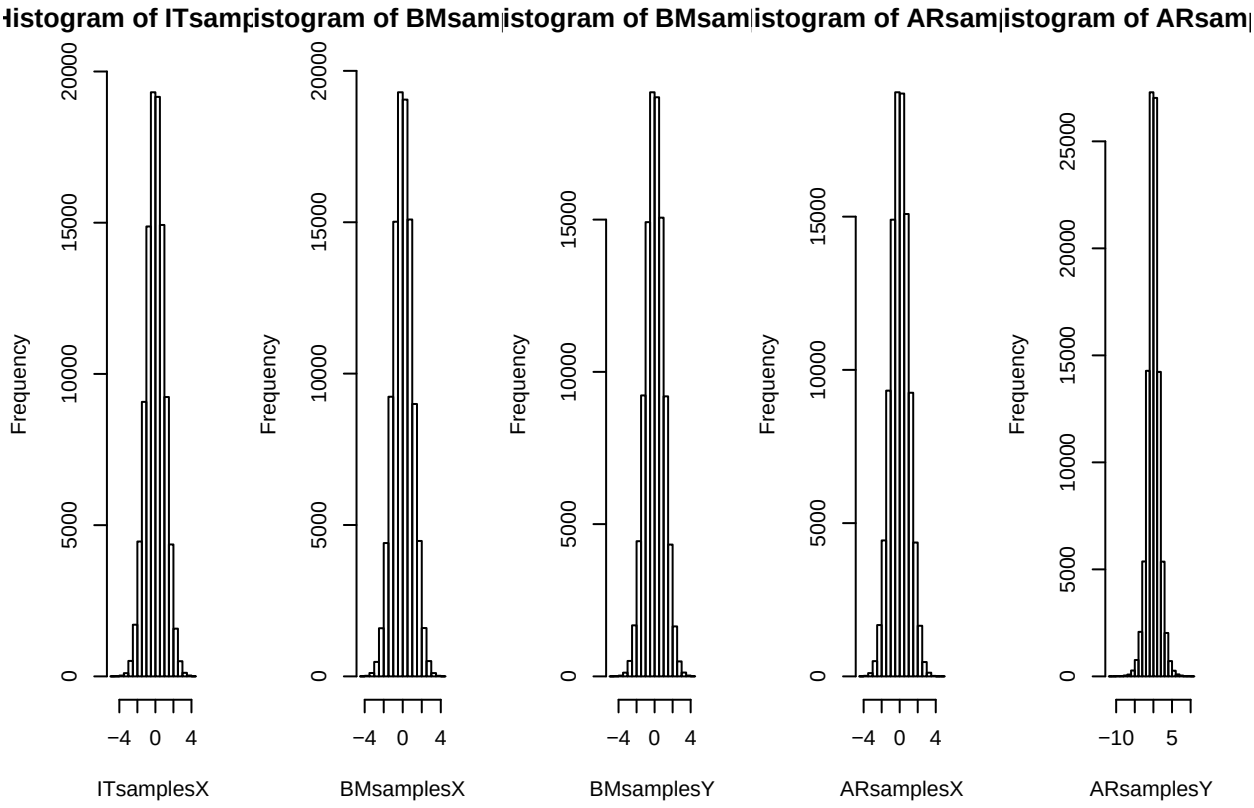

```
ggplot(ARdf) + geom_point(aes(x=N,y=elapsed,colour="green")) + ggtitle("AR time elapsed")
```



While running the three functions for the values of N, I found that 100,000 samples took a prohibitively long time to process. I scaled this down to 20,000 and ran each of the functions for $N = 100, 1000, 10000, 20000$. From the plots, we see that the processing time increases by a disproportionate magnitude between 10,000 and 20,000 samples. For each of the stats functions, we also see that the mean and standard deviation approaches 1 and 0, respectively, and varies within a few thousandths of a point at about 10,000 observations. Weighing between accuracy and efficiency, I would run all of these functions at N = 10,000. Of the three function, the Box-Müller (BM) approach seems to be the most accurate and show the least variability between each value of N.

Plot histograms of 100000 samples from each method and compare.

```
ITsamplesX <- c()
  for(i in 1:100000){ITsamplesX[i] <- normrandit()}
BMsamplesX <- c()
  for(i in 1:100000){BMsamplesX[i] <- normrandbm()[[1]]}
BMsamplesY <- c()
  for(i in 1:100000){BMsamplesY[i] <- normrandbm()[[2]]}
ARsamplesX <- c()
  for(i in 1:100000){ARsamplesX[i] <- normrandar()[[1]]}
ARsamplesY <- c()
  for(i in 1:100000){ARsamplesY[i] <- normrandar()[[2]]}
par(mfrow=c(1,5))
hist(ITsamplesX)
hist(BMsamplesX)
hist(BMsamplesY)
hist(ARsamplesX)
hist(ARsamplesY)
```



With the exception of the Y-values from the Acceptance-Rejection methods return a histogram that reflects a standard normal distribution.

Use Monte Carlo integration to estimate the value of π

In this exercise, we will create a Monte Carlo function that approximates the value of π by simulating the area under the curve of a quarter-circle, which is centered at $(0, 0)$ and has a radius equal to 1. To do so, our function will take one input, n, and generate two vectors of length n that simulate (x, y) coordinates between $0 \leq x \leq 1$ and $0 \leq y \leq 1$. The function will then calculate each value of $f(x_i)$ where $f(x) = \sqrt{1 - x^2}$. If $y_1 \leq f(x)_i$, we will add the value to a counter and then divide that counter by the total number of random points and multiply by 4, which should approximate the area circle given a relatively large number n.

```
estimatepi <- function(n){
  counter <- 0
  x <- runif(n,0,1)
  y <- runif(n,0,1)
  fx <- sqrt(1-x^2)
  for (i in 1:n){if (y[i] <= fx[i]){counter <- counter + 1}}
  area <- 4*counter/n
  se <- abs(area-pi)/sqrt(1) # taking the absolute value as a short cut to sd, instead of taking the sq
  CIupper <- area + se*1.96
  CIlower <- area - se*1.96
  return(c("estimate"=area,"standard error"=se,"upper limit of 95% CI"=CIupper,"lower limit of 95% CI"=C
                    }

N <- seq(1000,10000,by=500)
estimates <- matrix(data=NA, nrow=19, ncol=4)
rownames(estimates) <- N
colnames(estimates) <- names(estimatepi(1))
```

```
for(i in 1:length(N)){
    estimates[i,] <- estimatepi(N[i])
    }
estimates
```

```
##       estimate standard error upper limit of 95% CI lower limit of 95% CI
## 1000  3.148000   0.0064073464              3.160558              3.135442
## 1500  3.146667   0.0050740131              3.156612              3.136722
## 2000  3.164000   0.0224073464              3.207918              3.120082
## 2500  3.110400   0.0311926536              3.171538              3.049262
## 3000  3.132000   0.0095926536              3.150802              3.113198
## 3500  3.158857   0.0172644893              3.192696              3.125019
## 4000  3.198000   0.0564073464              3.308558              3.087442
## 4500  3.118222   0.0233704314              3.164028              3.072416
## 5000  3.146400   0.0048073464              3.155822              3.136978
## 5500  3.156364   0.0147709828              3.185315              3.127413
## 6000  3.141333   0.0002593203              3.141842              3.140825
## 6500  3.152615   0.0110227310              3.174220              3.131011
## 7000  3.161714   0.0201216321              3.201153              3.122276
## 7500  3.158933   0.0173406797              3.192921              3.124946
## 8000  3.161000   0.0194073464              3.199038              3.122962
## 8500  3.134118   0.0074750065              3.148769              3.119467
## 9000  3.124889   0.0167037647              3.157628              3.092150
## 9500  3.136421   0.0051716010              3.146557              3.126285
## 10000 3.135600   0.0059926536              3.147346              3.123854
```

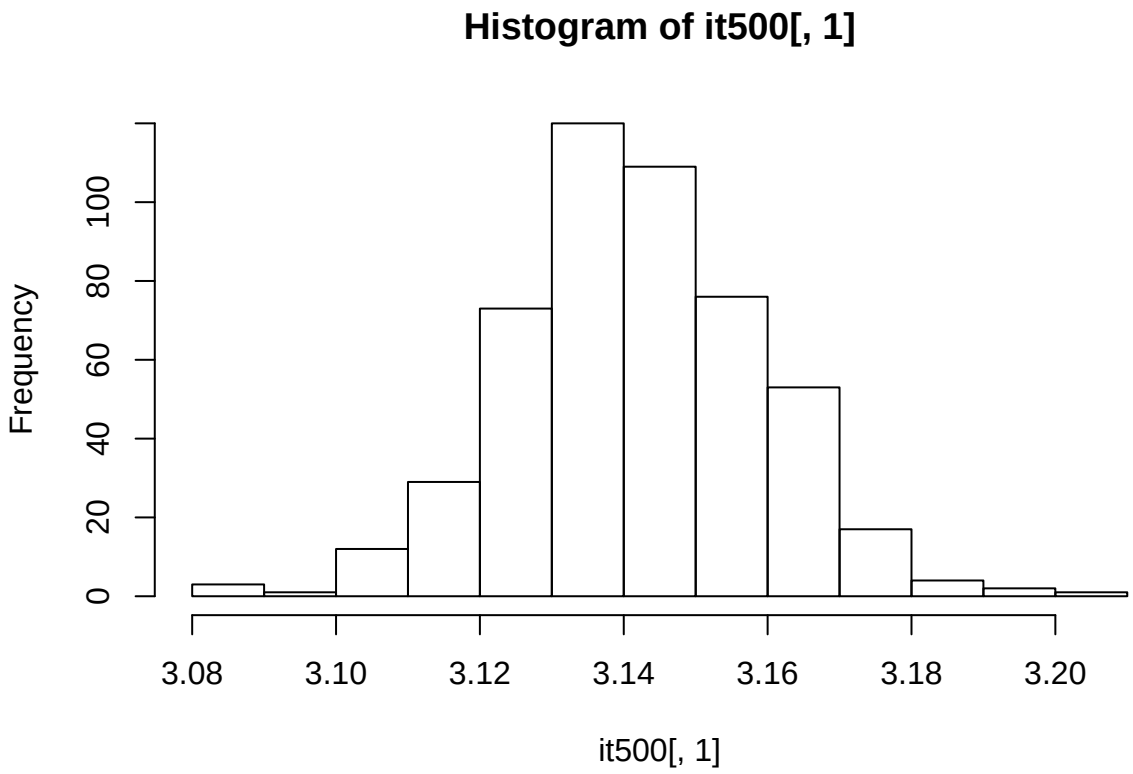
We can be 95% certain that our estimate will be within 0.1 of the true value of π at 1000 estimates. We can be 95% certain that our estimate will be within 0.01 of the true value of π at 8500 estimates. Now we will replicate estimatepi 500 times at N=8500.

```
it500 <- t(replicate(500,estimatepi(8500)))
head(it500)
```

```
##      estimate standard error upper limit of 95% CI lower limit of 95% CI
## [1,] 3.135529    0.006063242              3.147413              3.123645
## [2,] 3.144471    0.002877935              3.150111              3.138830
## [3,] 3.150588    0.008995582              3.168220              3.132957
## [4,] 3.132235    0.009357359              3.150576              3.113895
## [5,] 3.164235    0.022642641              3.208615              3.119856
## [6,] 3.145882    0.004289699              3.154290              3.137475
```

The histogram of the estimates seems appears to be normally distributed around the true value of π.

```
hist(it500[,1])
```



The standard error for N=8500 from our table above is 0.0001333005. It is far less (one-hundreth) than the standard deviation from the estimates:

```
sd(it500[,1])
```

```
## [1] 0.01720404
```

This will return the percentage of samples within the confidence interval.

```
counter <- 0
for (i in 1:500){
  if((estimates[16,3] >= it500[i,1]) & (estimates[16,4] <= it500[i,1])){
  counter <- counter + 1
  }
  inInterval <- counter/500
}
inInterval
```

```
## [1] 0.586
```