Homework 3

DATA604 Simulation and Modeling

Daniel Dittenhafer

March 6, 2016

1

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

```
fn1 <- function(x0)
{
  df <- data.frame(X=c(), R=c())
  x <- x0
  continue <- TRUE

  while(continue)
  {
    xi <- (11 * x) %% 16
    df <- rbind(df, data.frame(X=x, R=xi))
    x <- xi

    if(xi == x0)
    {
      break
    }
  }

  return(df)
}

res <- fn1(1)
```

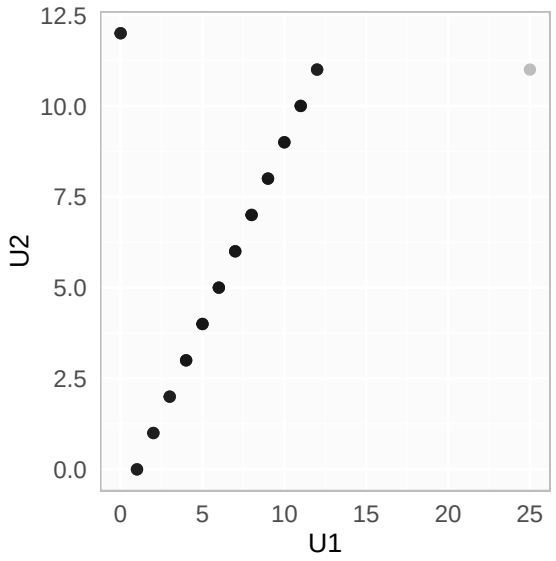
X	R
1	11
11	9
9	3
3	1

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 observe.

```
fn2 <- function(x0, max=100)
{
  df <- data.frame(U1=c(), U2=c())
  x <- x0

  for(i in 1:max)
  {
    xi <- (x + 12) %% 13
    df <- rbind(df, data.frame(U1=x, U2=xi))
```

```
    x <- xi
  }

  return(df)
}
# Call the function starting at x0=1
res <- fn2(25)
```



The chart above suggests there are only 13 points, but actually the LCG cycle period is 13 and numbers are repeating.

U1	U2
25	11
11	10
10	9
9	8
8	7
7	6
6	5
5	4
4	3
3	2
2	1
1	0
0	12
12	11

3

Implement the pseudo-random number generator:

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

Using the seed 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 observvations to see if they are independent.

```
fnLCG3 <- function(seed = 1, n = 1)
{
```

```
  rands <- rep(NA, n)
  x <- seed
  modVal <- (2^31 - 1)

  for(i in 1:n)
  {
    xi <- (16807 * x) %% (modVal)
    rands[i] <- xi
    x <- xi
  }

  return(rands)
}

n=100000
rn <- fnLCG3(1234567, n)
```

The first 6 generated numbers are shown below:

1422014746
456328559
849987676
681650688
1825340118
1687465831

Chi-Square Test

```
intervals <- 20
maxRn <- max(rn)
minRn <- min(rn)
intWidth <- (maxRn - minRn) / intervals
lwr <- minRn
dfCounts <- data.frame(intID=c(), count=c())
# Bin the data ourselves, I'd guess there
# is an easier way, but this will do.
for(i in 1:intervals)
{
  upr <- lwr + intWidth
  inRange <- rn[lwr <= rn & rn < upr]
  dfCounts <- rbind(dfCounts, data.frame(intID=i, count=length(inRange)))
  # setup for next interval range
  lwr <- upr
}
# Do our own Chi-Squared test
Expected <- (100000 / intervals)
chi2 <- sum((dfCounts$count - Expected)^2 / Expected)
chi2
```

```
## [1] 14.7762
```

```
# Use built-in function to compare
chiTest <- chisq.test(dfCounts$count)
chiTest
```

```
##
```

```
##  Chi-squared test for given probabilities
##
## data:  dfCounts$count
## X-squared = 14.776, df = 19, p-value = 0.7367
```

The p-value = 0.7367029 is not less than $\alpha = 0.05$, therefore we don't reject the null hypothesis that the distrubtion is uniform.

intID	count
1	5069
2	5028
3	5044
4	5087
5	4948
6	4953
7	4937
8	4933
9	4900
10	4957
11	5088
12	4994
13	5076
14	5019
15	5002
16	5067
17	4981
18	4914
19	5062
20	4940

Runs Up-and-Down Test

Using the `tseries` package, we execute the Runs test (Trapletti and Hornik, 2015). First we have to construct the +/- vector. Here we simply convert to boolean.

```
s <- rep(NA, n - 1)
for(i in 1:n - 1)
{
  s[i] <- rn[i] < rn[i + 1]
}

runsTest <- runs.test(as.factor(s))
runsTest
```

```
##
##  Runs Test
##
## data:  as.factor(s)
## Standard Normal = 105.84, p-value < 2.2e-16
## alternative hypothesis: two.sided
```

Based on the p-value < 0.05, we reject the null hypothesis and conclude there is not evidence to support independence in the psuedo-random numbers.

4.

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

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

Inverse-Transforms

First find the indefinite integral of the probability density function:

$$F(x) = \int \frac{3x^2}{2} dx = \frac{x^3}{2}$$

Next set $F(x) = R$ and solve for x in terms of R:

$$\frac{x^3}{2} = R$$

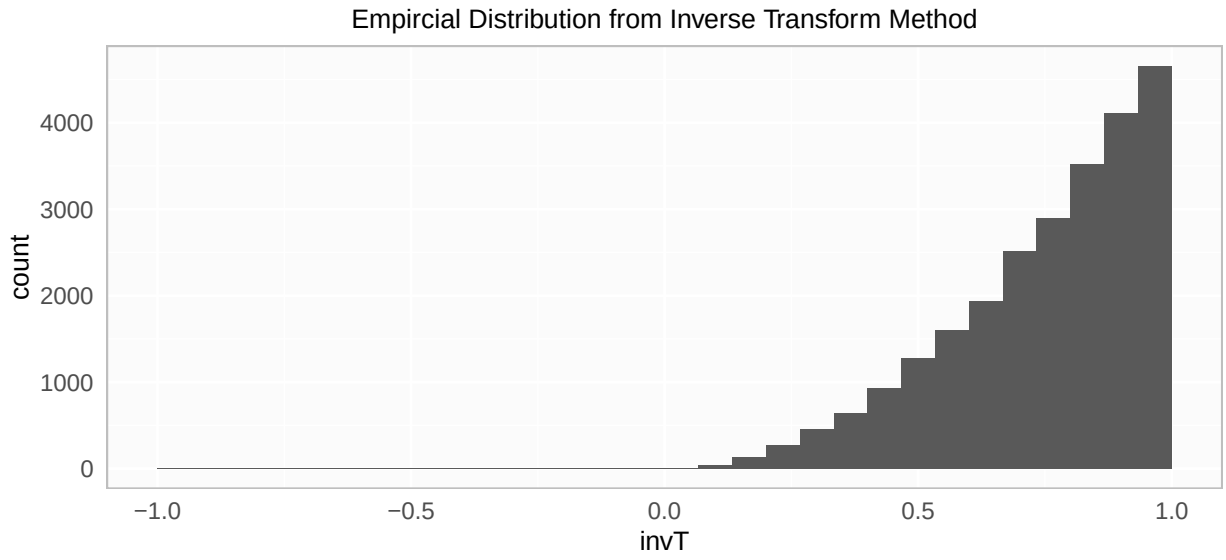
$$x^3 = 2R$$

$$F^-1(x) = \sqrt[3]{2R}$$

```
# Define a function of the F^-1(X)
invTfn4 <- function(r)
{
  vals <- (2 * r)^(1/3)
  return (vals)
}

# Generate the uniform psuedo-random vars
rVals <- runif(n, -1, 1)
# Convert to the desired distribution using the inverse transform method.
invTVals <- invTfn4(rVals)
```

```
## Warning: Removed 74997 rows containing non-finite values (stat_bin).
```

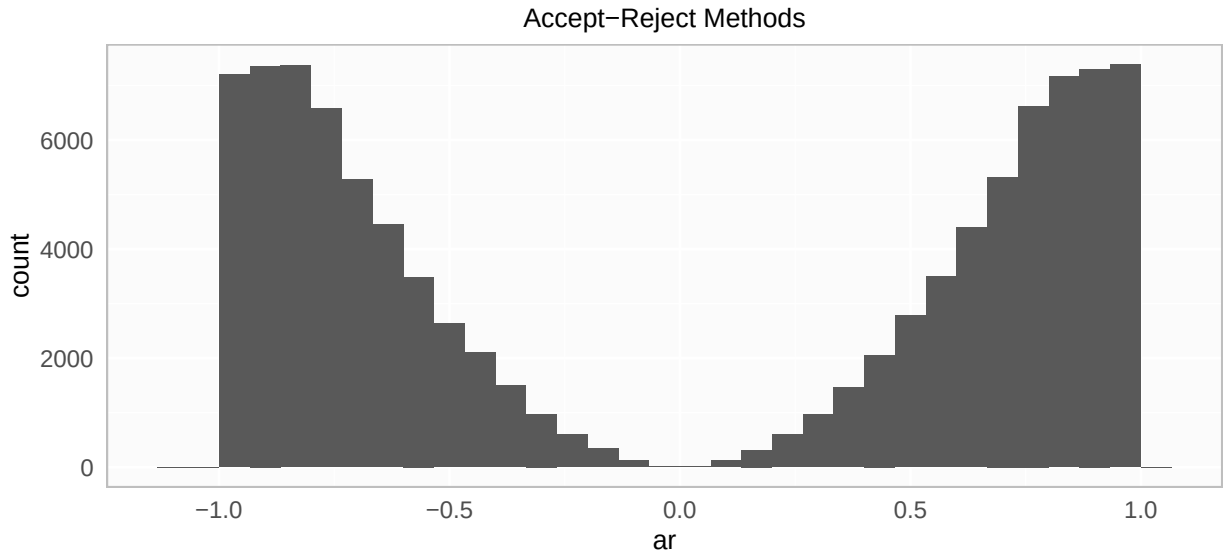
```
## Warning: Removed 2 rows containing missing values (geom_bar).
```



Acceptance-Rejection Method

```
# Helper function for the Accept/Reject approach
myRARmethod <- function(fun, min, max)
{
  M <- 2
  accepted <- FALSE
  while(!accepted)
  {
    # Get a random value from uniform distrubtion (g(x) for us)
    r <- runif(1, min, max)

    # Sample x from g(x) and u from U(0,1) (the uniform distribution over the unit interval)
    u <- runif(1, 0, 1)
    gx <- dunif(r, min, max)

    # Check whether or not u<f(x)/Mg(x).
    if(u < fun(r) / (M * gx))
    {
      accepted = TRUE
    }
  }

  return(r)
}

# Define a function for the PDF
Arfn4 <- function(x)
{
  if(-1 <=x && x <= 1)
  {
    val <- (3 * x^2) / 2
  }
  else
  {
    val = 0
  }
  return (val)
}

# Loop to generate the values
rarVals <- rep(NA, n)
for(i in 1:n)
{
  rarVals[i] <- myRARmethod(Arfn4, -1, 1)
}
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Composition

Hmm....

5

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

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

Inverse Transform Method

$$F(X) = \int \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}} dx$$

It turns out the `qnorm` function is the inverse normal CDF function, so we'll use it.

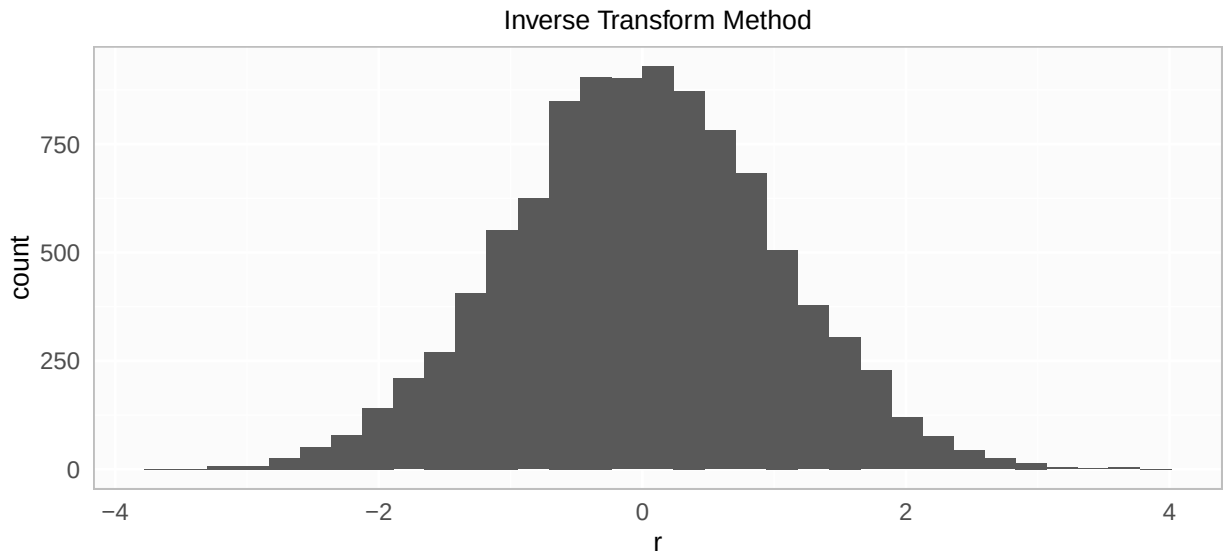
```
normrandit <- function()
{
  u <- runif(1)
  return (qnorm(u))
}

itstats <- function(n)
{
  vals <- rep(NA, n)
  for(i in 1:n)
  {
    vals[i] <- normrandit()
  }

  return(list(values=vals, mean=mean(vals), sd=sd(vals)))
}

res <- itstats(10000)
res$mean
```

```
## [1] -0.005543553
```

```
res$sd
```

```
## [1] 1.003801
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Box-Muller Method

```
normrandbm <- function()
{
  u2 <- runif(2)

  v1 <- (-2 * log(u2[1]))^(1/2) * cos(2 * pi * u2[2])
  v2 <- (-2 * log(u2[1]))^(1/2) * sin(2 * pi * u2[2])

  return (c(v1, v2))
}

bmstats <- function(n)
{
  vals <- rep(NA, n)
  for(i in seq(1, n, by=2))
  {
    rs <- normrandbm()
    vals[i] <- rs[1]
    vals[i + 1] <- rs[2]
  }

  return(list(values=vals, mean=mean(vals), sd=sd(vals)))
}

resBm <- bmstats(10000)
length(resBm[[1]])
```

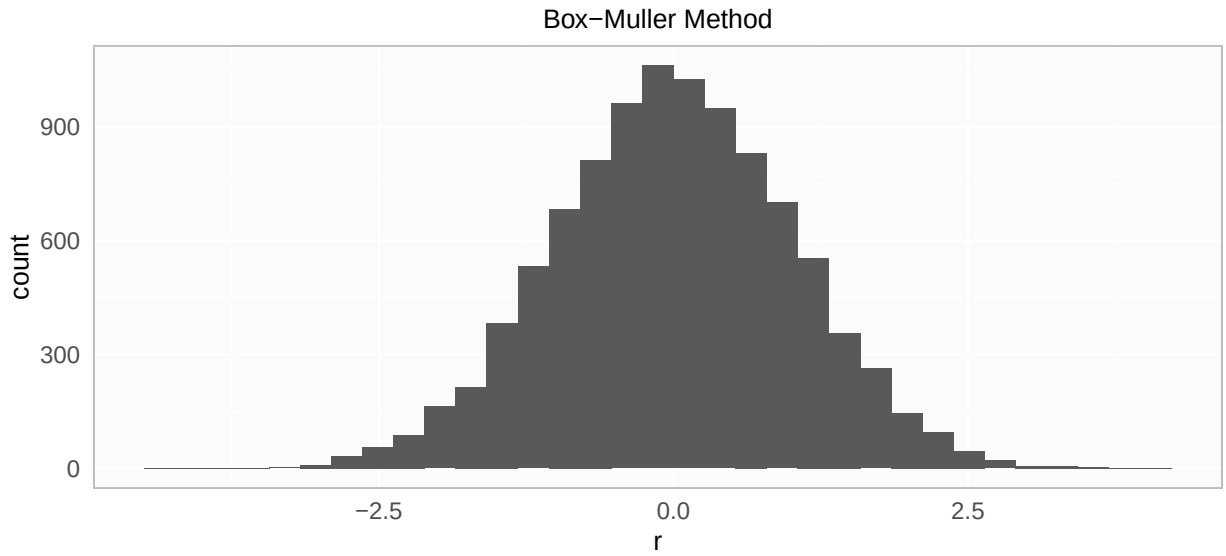
```
## [1] 10000
```

```
# Mean
resBm$mean
```

```
## [1] 0.00486827
```

```
# SD
resBm$sd
```

```
## [1] 1.009415
```



Accept Reject Method

```
normrandar <- function()
{
  continue <- TRUE
  while(continue)
  {
    u2 <- runif(2)
    eu2 <-  - log(u2)

    if(eu2[2] >= ( (eu2[1] - 1)^2 / 2) )
    {
      break
    }
  }

  sign <- runif(1)
  if(sign > 0.5)
  {
    eu2[1] <- eu2[1] * -1
  }

  return(eu2[1])
}

arstats <- function(n)
{
```

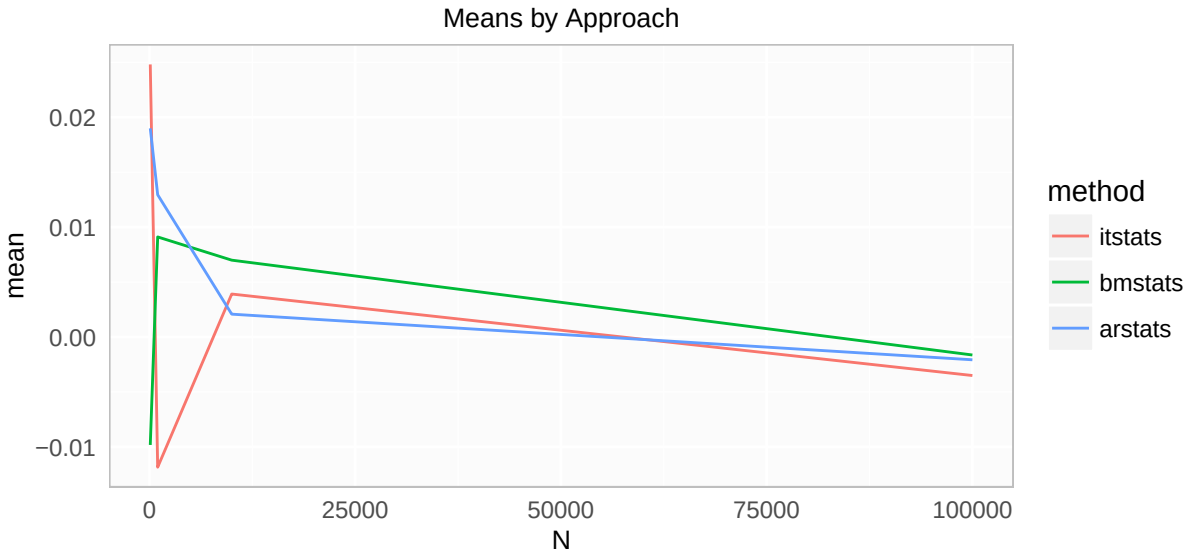
```
  vals <- rep(NA, n)
  for(i in seq(1, n))
  {
    rs <- normrandar()
    vals[i] <- rs
    #vals[i + 1] <- rs[2]
  }

  return(list(values=vals, mean=mean(vals), sd=sd(vals)))
}
```

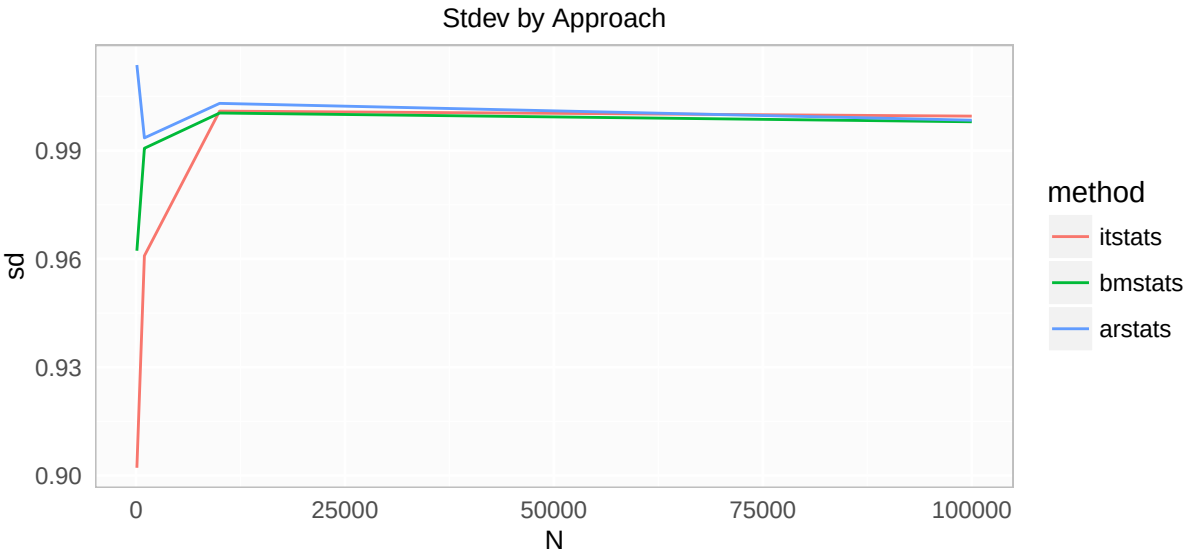
Compare

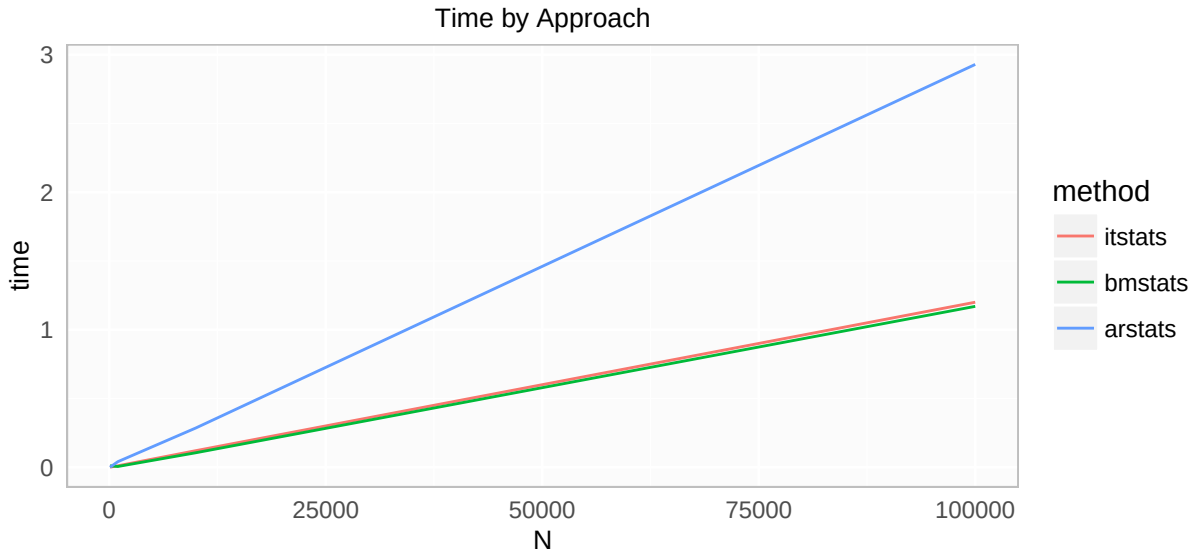
```
dfCompare <- data.frame(method=c(), N=c(), mean=c(), sd=c(), time=c())

perfTest <- function(fun)
{
  dfResult <- data.frame(method=c(), N=c(), mean=c(), sd=c(), time=c())
  Ns <- c(100, 1000, 10000, 100000)
  it <- 2

  for(n in Ns)
  {
    m <- rep(NA, it)
    s <- rep(NA, it)
    t <- rep(NA, it)
    for(i in 1:it)
    {
      st <- system.time({ret <-  fun(n)})
      #print(st)
      m[i] <- ret$mean
      s[i] <- ret$sd
      t[i] <- st[[3]]
    }

    dfResult <- rbind(dfResult, data.frame(method=deparse(substitute(fun)),
                                           N=n,
                                           mean=mean(m),
                                           sd=mean(s),
                                           time=mean(t)))
  }

  return (dfResult)
}
# Force less use of scientific notation
# http://stackoverflow.com/questions/9397664/force-r-not-to-use-exponential-notation-e-g-e10
options("scipen"=100, "digits"=5)

dfCompare <- rbind(dfCompare, perfTest(itstats))
dfCompare <- rbind(dfCompare, perfTest(bmstats))
dfCompare <- rbind(dfCompare, perfTest(arstats))
```

method	N	mean	sd	time
itstats	100	0.02481	0.90215	0.000
itstats	1000	-0.01185	0.96086	0.010
itstats	10000	0.00391	1.00094	0.120
itstats	100000	-0.00350	0.99954	1.200

method	N	mean	sd	time
bmstats	100	-0.00982	0.96226	0.010
bmstats	1000	0.00911	0.99064	0.005
bmstats	10000	0.00700	1.00040	0.105
bmstats	100000	-0.00163	0.99798	1.170
arstats	100	0.01899	1.01370	0.000
arstats	1000	0.01295	0.99351	0.040
arstats	10000	0.00208	1.00308	0.285
arstats	100000	-0.00207	0.99842	2.930



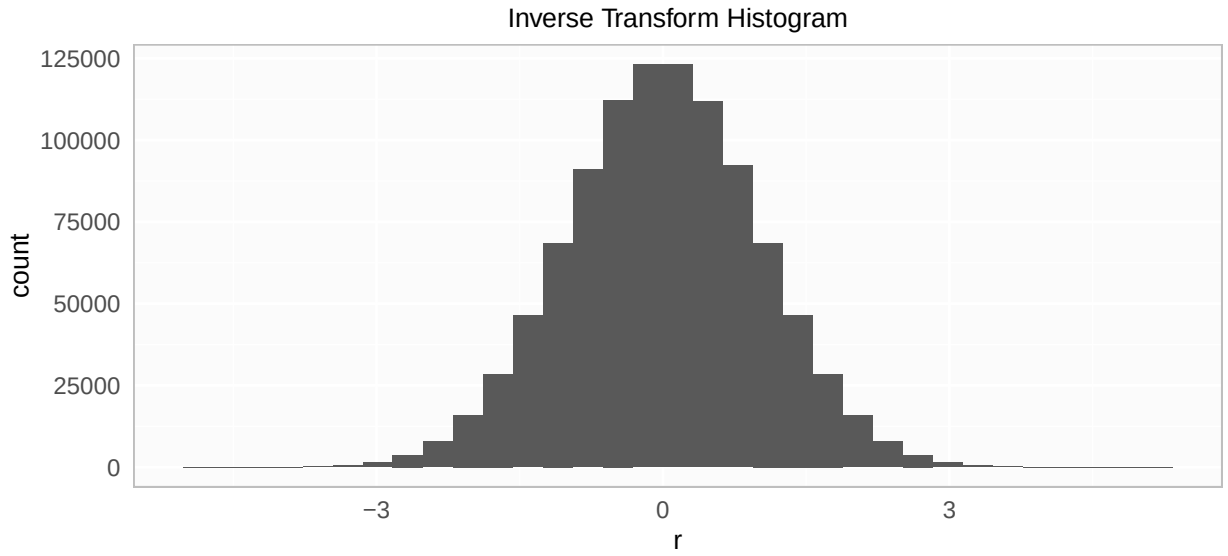


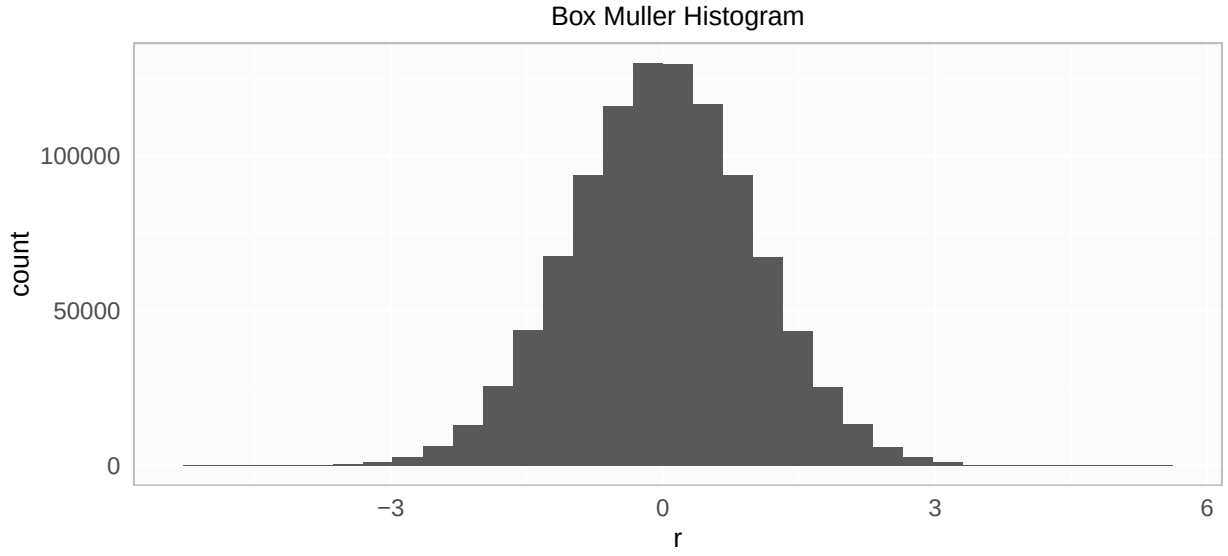


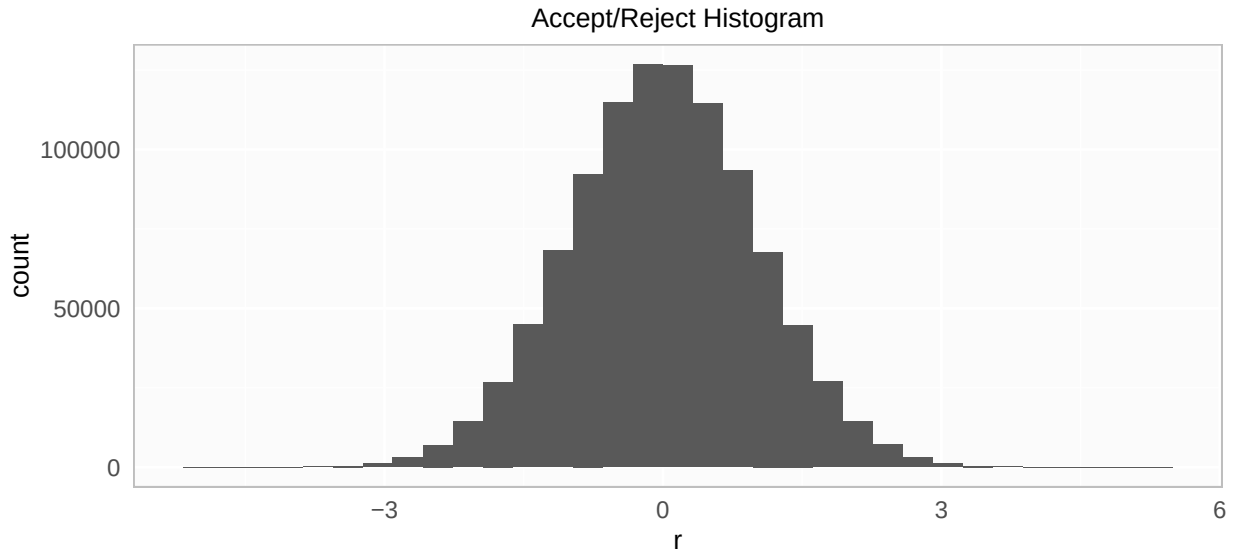
For me, the Box-Muller approach takes the least time with Inverse Transform close behind. Also, I found that preallocating the vector in the stats method had a significant effect on performance. If the vector is not fully allocated, the time for the 100K samples jumps to 30+ seconds.

Million Sample Histograms

```
filename <- "MillionSamples.RData"
if(!file.exists(filename))
{
  m <- 1000000
  itMil <- itstats(m)
  bmMil <- bmstats(m)
  arMil <- arstats(m)

  dfItStats <- data.frame(method=rep("itstats", m), r=itMil$values)
  dfBmStats <- data.frame(method=rep("bmstats", m), r=bmMil$values)
  dfArStats <- data.frame(method=rep("arstats", m), r=arMil$values)

  #dfAll$method <- as.factor(dfAll$method)

  #save(dfAll, file=filename)
} else
{
  load(filename)
}
```







6

Use Monte Carlo integration to estimate the value of π.

```r
# Define a function of the circle
quarterCircleY <- function(x)
{
  val <- sqrt((-1 * x^2) + 1)
  return (val)
}

insidecircle <- function(x, y)
{
  yq <- quarterCircleY(x)
  inside <- y <= yq

  return (inside)
}

# Define function for the Monte Carlo iterations
estimatepi <- function(n, mc=1)
{
  #mc <- 100
  dfMcPi <- data.frame()
  for(i in 1:mc)
  {
    #dfMcPoints <- monteCarloPiEst(n_points)
    dfPoints <- data.frame()
    x <- runif(n)
    y <- runif(n)

    inside <- insidecircle(x, y)

    dfPoints <- data.frame(x, y, inside)

    #return (dfPoints)

    # Compute quarter circle area under the curve
    qca <- sum(dfPoints$inside) / nrow(dfPoints)
    qca
    # estimate Pi
    pi_est <- qca * 4

    # Add to our running list.
    dfMcPi <- rbind(dfMcPi, cbind(i, pi_est))
  }
  mnPi <- mean(dfMcPi$pi_est)
  se <- sd(dfMcPi$pi_est) / sqrt(n)
  ci95 <- c(mnPi - (se * 1.96), mnPi + (se * 1.96))

  return(list(pi_est=mnPi, se=se, ci95=ci95, mc_results=dfMcPi, points=dfPoints))
}


dfRes <- data.frame(pi_est=c(), se=c(), lwr=c(), upr=c(), n=c())
for(ni in seq(1000, 10000, by=500))
{
  resPi <- estimatepi(ni, mc=100)
  dfRes <- rbind(dfRes, data.frame(pi_est=resPi$pi_est,
                                   se=resPi$se,
                                   lwr=resPi$ci95[1],
                                   upr=resPi$ci95[2], n=ni))
}
```

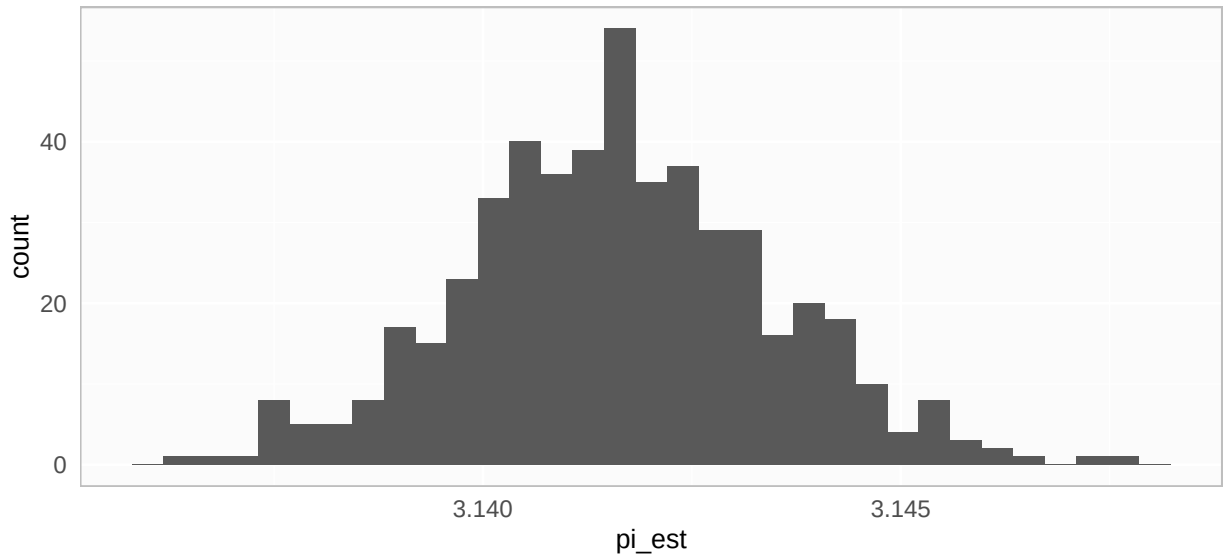
```
kable(dfRes)
```

pi_est	se	lwr	upr	n
3.1449	0.00167	3.1416	3.1482	1000
3.1485	0.00126	3.1461	3.1510	1500
3.1433	0.00082	3.1417	3.1449	2000
3.1419	0.00068	3.1406	3.1432	2500
3.1373	0.00057	3.1362	3.1384	3000
3.1401	0.00041	3.1393	3.1409	3500
3.1389	0.00038	3.1382	3.1397	4000
3.1436	0.00038	3.1429	3.1444	4500
3.1433	0.00030	3.1427	3.1439	5000
3.1434	0.00030	3.1428	3.1440	5500
3.1385	0.00029	3.1380	3.1391	6000
3.1425	0.00024	3.1420	3.1430	6500
3.1420	0.00022	3.1416	3.1425	7000
3.1437	0.00019	3.1433	3.1440	7500
3.1390	0.00022	3.1386	3.1394	8000
3.1419	0.00022	3.1415	3.1423	8500
3.1411	0.00019	3.1407	3.1415	9000
3.1430	0.00017	3.1426	3.1433	9500
3.1434	0.00017	3.1431	3.1437	10000

Using n=8500:

```
n=8500
dfRes2 <- data.frame(pi_est=c(), se=c(), lwr=c(), upr=c(), n=c())
for(k in 1:500)
{
  resPi <-estimatepi(n, mc=100)
  dfRes2 <- rbind(dfRes2, data.frame(pi_est=resPi$pi_est, se=resPi$se, lwr=resPi$ci95[1], upr=resPi$ci95[2], n
}
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
theRow <- dfRes[dfRes$n == n,]
```

```
sdEstPi <- sd(dfRes2$pi_est)
sdEstPi
```

```
## [1] 0.0018386
```

Standard deviation does not quite match the SE from earlier (0.00022).

```
subCI <- dfRes2$pi_est[theRow$lwr <= dfRes2$pi_est & dfRes2$pi_est <= theRow$upr]

prop95 <- length(subCI) / 500
prop95
```

```
## [1] 0.204
```

References

Trapletti, A. and K. Hornik. tseries: Time Series Analysis and Computational Finance. R package version 0.10-34. 2015. URL: http://CRAN.R-project.org/package=tseries.