STAT 242 Final Project: Adaptive Rejection Sampling

Jonathan Morrell, Ziyang Zhou, Vincent Zijlmans

December 12, 2018

In this project we implement the adaptive rejection sampling algorithm proposed by Gilks et al. (1995), a method for rejection sampling from any univariate log-concave probability density function $f(x)$. This method is particularly useful in situations where the density function is computationally expensive to evaluate, as it uses successive evaluations of $h(x) = \log(f(x))$ to improve the approximation of a rejection envelope $u(x)$ and squeezing function $l(x)$, which are (theoretically) much simpler to compute and sample from than the original density function.

The R package, `ars`, containing our solution can be found using the following link:

https://github.com/jtmorrell/ars

Or, can be installed using devtools:

```
devtools::install_github('jtmorrell/ars')
```

Methodology

Our implementation consists of one function, `ars`, which takes as arguments the univariate log-concave density function $f(x)$, the number of samples desired N, the initial points x_0 used to approximate $h(x) = \log(f(x))$ (default: `c(-1, 1)`), and optionally the bounds of the density function.

To demonstrate the implementation of `ars` we will go through the example of sampling 100000 points from the normal distribution.

```
## Normally inputs to ars()
f <- dnorm
N <- 100000
x0 <- c(-1.0, 1.0)
bounds <- c(-Inf, Inf)
```

First we check the inputs.

```
## Input sanity check
if (!is.function(f)) {
  stop('f is not a function.')
}

if (length(bounds) != 2) {
  stop('Length of bounds must be 2! (upper and lower bounds required)')
}

if (bounds[1] == bounds[2]) {
  warning('Same upper and lower bounds, replacing them by default: bounds=c(-Inf, Inf)')
  bounds <- c(-Inf, Inf)
}

if (!all((x0 > bounds[1]) & (x0 < bounds[2]))) {
  stop('Bad inputs: x0 must be inside bounds.')
}
```

Then we'll initialize some helper variables.

```
## define h(x) = log(f(x))
h <- function(x) {
  return (log(f(x)))
}

## helper variables
dx <- 1E-8  # mesh size for finite difference approximation to h'(x)
eps <- 1E-7  # tolerance for non-log-concavity test
max_iters <- 10000  # prevent infinite loop
current_iter <- 0
bds_warn <- FALSE  # Flag for boundary warning in main while loop
```

Because the derivative of the log-density function $h'(x)$ is required in our calculation, we need to approximate it using a finite-difference method. We'll use a forward differencing approximation, as we only need to evaluate $h(x)$ one additional time to calculate $h'(x)$. The forward difference method uses the following Taylor expansion

$$f(x_0 + h) = f(x_0) + f'(x_0)h + \frac{1}{2}f''(x_0)h^2 + \mathcal{O}(h^3)$$

to approximate the first derivative as

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

where h is a small number, often called the mesh spacing in numerical methods.

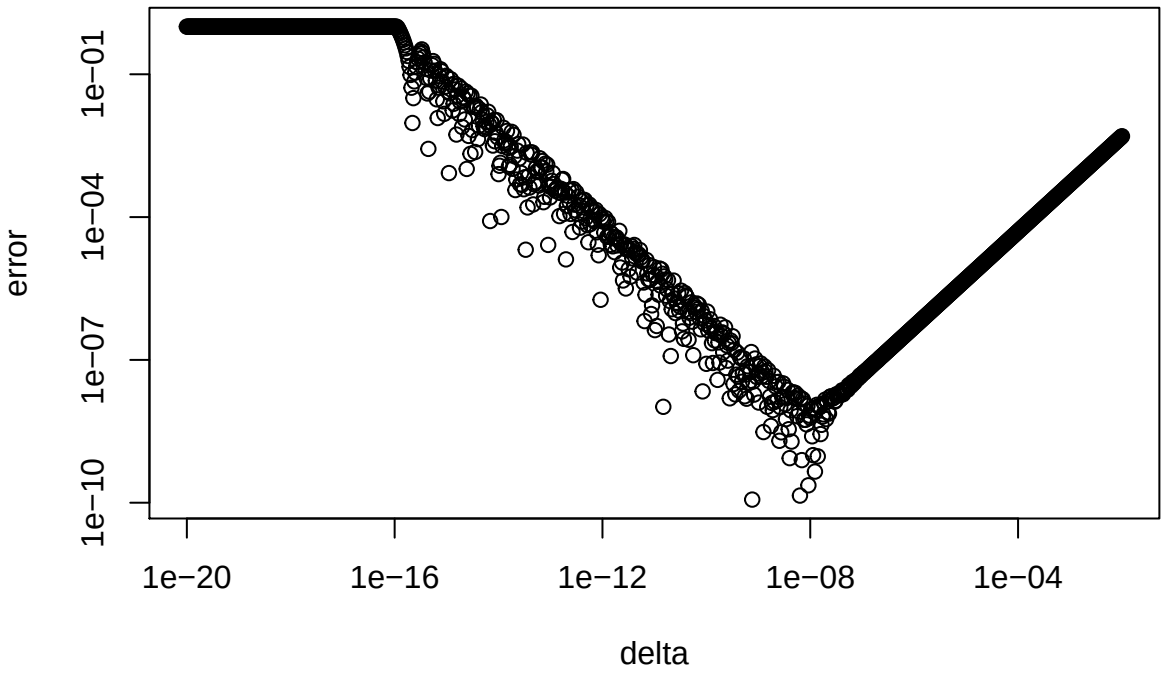
This approximation ignores higher order corrections, which means it will have first order truncation error:

$$\epsilon = \frac{1}{2}f''(x_0)h + \mathcal{O}(h^2)$$

Therefore, we want to choose our mesh spacing h as small as possible. However, because we are performing these computations on double-precision floating point computers there is a limit to how small we can make h before machine precision impacts the precision of this approximation. Looking at how we will compute the approximation of the derivative, we see that the numerator $f(x_0 + h) - f(x_0)$ requires the subtraction of two very similar numbers. Assuming $\mathcal{O}(f'(x)) \approx 1$, the difference will be $\approx \mathcal{O}(h)$. However the numbers we're subtracting have 16 digits of accuracy, so if h were 10^{-10} we the difference would only have 6 digits of accuracy! (The numerator and denominator are of similar order, so we expect only a small loss of precision in the division operation). Therefore we expect the mesh spacing with the highest accuracy to be around 10^{-8}.

If we consider the simple example of $f(x) = x^2$ we can see that this is approximately correct.

```
## vector of possible mesh spacings
delta <- 10**seq(-20, -2, length = 1000)
## example function x^2
quad <- function(x) x**2
## approximate derivative using forward difference
h_prime <- (quad(1.0 + delta) - quad(1.0))/delta
## Difference from true value of 2
error <- abs(h_prime - 2.0)/2.0
plot(delta, error, log = 'xy')
```



Continuing on with the ARS algorithm

```
## initialize vectors
x0 <- sort(x0)  # Ensures x0 is in ascending order
x_j <- c()  # evaluated x values
h_j <- c()  # evaluated h(x) values
dh_j <- c()  # evaluated h'(x) values
x <- c()  # accepted sample vector

## Evaluate h(x0)
h0 <- h(x0)
## Finite difference approximation to h'(x)
dh0 <- (h(x0 + dx) - h0)/dx

## Check for NaNs and infinities
isnum <- is.finite(h0)&is.finite(dh0)
x0 <- x0[isnum]
h0 <- h0[isnum]
dh0 <- dh0[isnum]

## Error if no good x0 values
if (length(h0) == 0) {
  stop('h(x0) either infinite or NaN.')
}


## ARS requires either finite bounds or at least
## one positive and one negative derivative
if (!(dh0[1] > 0 || is.finite(bounds[1]))) {
  stop('dh(x0)/dx must have at least one positive value, or left bound must be greater than -infinity.')
}

if (!(dh0[length(dh0)] < 0 || is.finite(bounds[2]))) {
```

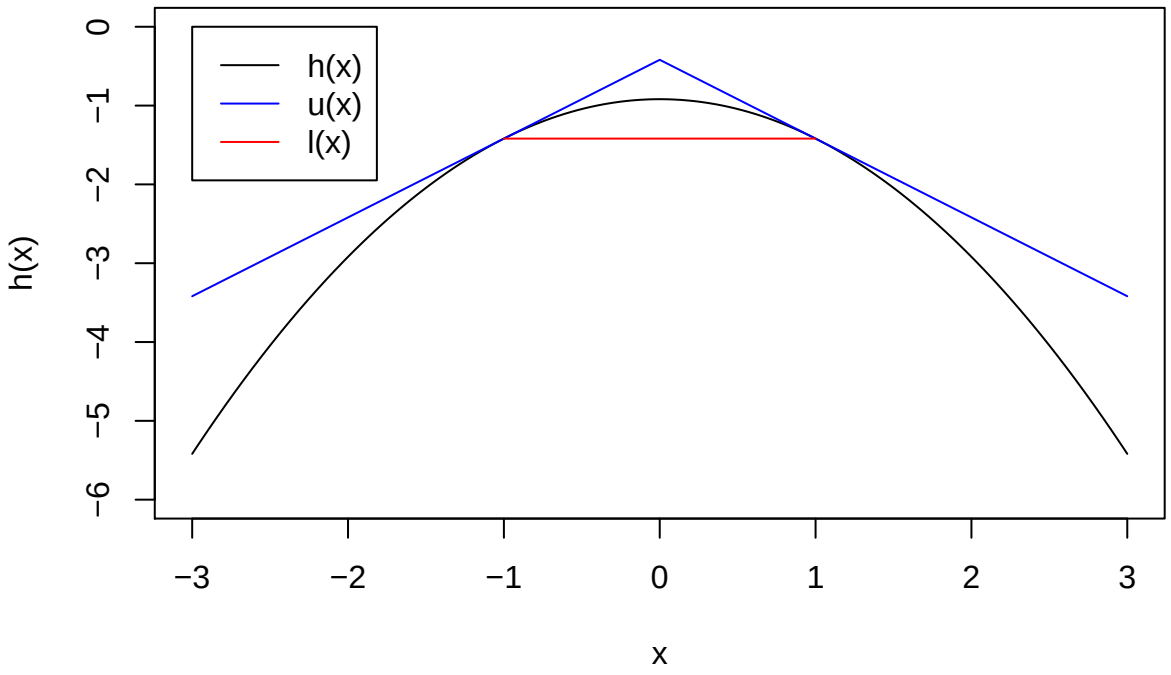
```
  stop('dh(x0)/dx must have at least one negative value, or right bound must be less than infinity.')
}
```

The first step specified by Wilks is to initialize our functions $u_k(x)$, $l_k(x)$, $s_k(x)$ and to pre-compute the intercepts of $u_k(x)$ (z_k). To do this we will write helper functions to calculate the coefficients of the piecewise linear functions as specified by Wilks.

```
##### HELPER FUNCTIONS #####

calc_z_j <- function (x, h, dh, bounds) {
  ## Calculate intercepts of piecewise u_j(x)
  ##
  ## Arguments
  ## x, h, dh: evaluated x, h(x) and h'(x)
  ## bounds: bounds of distribution h(x)
  ##
  ## Value
  ## intercepts (bounds) of piecewise u_j(x)

  L <- length(h)
  z <- (h[2:L] - h[1:L-1] - x[2:L]*dh[2:L] + x[1:L-1]*dh[1:L-1])/(dh[1:L-1] - dh[2:L])

  return (append(bounds[1], append(z, bounds[2])))
}

calc_u <- function (x, h, dh) {
  ## Calculate slope/intercept for piecewise u_j(x)
  ##
  ## Arguments
  ## x, h, dh: evaluated x, h(x), and h'(x)
  ##
  ## Value (list)
  ## m, b: slope/intercept of u(x)

  return (list(m = dh, b = h - x*dh))
}

calc_l <- function (x, h) {
  ## Calculate slope/intercept for piecewise l_j(x)
  ##
  ## Arguments
  ## x, h: evaluated x, and h(x)
  ##
  ## Value (list)
  ## m, b: slope/intercept of l(x)

  L <- length(h)
  m <- (h[2:L] - h[1:L-1])/(x[2:L] - x[1:L-1])
  b <- (x[2:L]*h[1:L-1] - x[1:L-1]*h[2:L])/(x[2:L] - x[1:L-1])

  return (list(m = m, b = b))
}
```

We see that already with only two points we have a good approximation to the rejection envelope $u_k(x)$ and squeezing function $l_k(x)$.

```r
u <- calc_u(x0, h0, dh0)
u <- c(-3*u$m[1] + u$b[1], u$b[1], 3*u$m[2] + u$b[2])

l <- calc_l(x0, h0)
l <- l$m*x0 + l$b

x <- seq(-3, 3, length = 1000)
plot(x, h(x), ylim = c(-6, 0), type = 'l', col = 'black')

lines(c(-3, 0, 3), u, col = 'blue')
lines(x0, l, col = 'red')
legend(-3, 0, legend = c('h(x)', 'u(x)', 'l(x)'),
       col = c('black', 'blue', 'red'), lty = 1)
```



For the function

$$s_k(x) = \exp u_k(x) / \int_D \exp u(x')dx'$$

from which we'll sample x, instead of computing the function we will compute coefficients which will allow us to sample easily. To do this we'll re-define $s_k(x)$ as

$$s_k(x) = e^{u_k(x)} / \int_D e^{u(x')}dx' = \frac{e^{b_k}e^{m_k x}}{\sum_{j=1}^K \int_{z_j}^{z_{j+1}} e^{b_j}e^{m_j x'}dx'} = \frac{e^{b_k}e^{m_k x}}{\sum_{j=1}^K \frac{e^{b_j}}{m_j}(e^{m_j z_{j+1}} - e^{m_j z_j})} \equiv \beta_k e^{m_k x}$$

for $m_j \neq 0$, or

$$s_k(x) = \frac{e^{b_k}e^{m_k x}}{\sum_{j=1}^K e^{b_j}(z_{j+1} - z_j)} \equiv \beta_k e^{m_k x}$$

where $m_j = 0$.

To sample x, we will first use weighted random sampling to determine which segment to sample from, where the weights used are the area of each segment

$$w_j = \int_{z_j}^{z_{j+1}} \beta_j e^{m_j x'} dx' = \frac{\beta_j}{m_j}(e^{m_j z_{j+1}} - e^{m_j z_j})$$

where $m_j \neq 0$, or

$$w_j = \beta_j(z_{j+1} - z_j)$$

where $m_j = 0$.

Then we will use inverse CDF sampling to sample x from segment j:

$$S_j(x) = \mathcal{U}(0,1) = u = \frac{1}{w_j}\int_{z_j}^{x} s_j(x')dx' = \frac{\beta_j}{w_j m_j}(e^{m_j x} - e^{m_j z_j})$$

giving us

$$x = \frac{1}{m_j}\log(\frac{w_j m_j}{\beta_j}u + e^{m_j z_j})$$

for $m_j \neq 0$, and $x = z_j + \frac{w_j}{\beta_j}u$ for $m_j = 0$.

```
calc_s_j <- function (m, b, z) {
  ## Calculate beta values and weights
  ##   needed to sample from u(x)
  ##
  ## Arguments
  ## m, b: slope/offset for piecewise u_j(x)
  ## z: bounds z_j for piecewise function u_j(x)
  ##
  ## Value (list)
  ## beta: amplitude of each piecewise segment in u(x)=beta*exp(m*x)
  ## w: area of each segment

  L <- length(z)
  eb <- exp(b)  # un-normalized betas
  eb <- ifelse(is.finite(eb), eb, 0.0)  # Inf/Nan -> 0

  ## treat m=0 case
  nz <- (m != 0.0)
  nz_sum <- sum((eb[nz]/m[nz])*(exp(m[nz]*z[2:L][nz]) - exp(m[nz]*z[1:L-1][nz])))
  z_sum <- sum(eb[!nz]*(z[2:L][!nz] - z[1:L-1]))

  ## Ensure integral of s(x) > 0
  if (nz_sum + z_sum <= 0.0) {
    stop('Area of s(x)=exp(u(x)) <= 0')
  }

  ## Normalize beta
  beta <- eb/(nz_sum + z_sum)
  ## Calculate weights for both m!=0 and m=0 cases
  w <- ifelse(nz, (beta/m)*(exp(m*z[2:L]) - exp(m*z[1:L-1])), beta*(z[2:L] - z[1:L-1]))
```

```
  return (list(beta = beta, w = ifelse((w > 0) & is.finite(w), w, 0.0)))
}

draw_x <- function (N, beta, m, w, z) {
  ## Sample from distribution u(x), by selecting segment j
  ##   based on segment areas (probabilities), and then sample
  ##   from exponential distribution within segment using
  ##   inverse CDF sampling.
  ##
  ## Arguments
  ## N: number of samples
  ## beta, m, w: parameters of u_j(x)
  ## z: intercepts (bounds) of u_j(x)
  ##
  ## Value (list)
  ## x: samples from u(x)
  ## J: segment index j of each sample

  J <- sample(length(w), N, replace = TRUE, prob = w)  # choose random segments with probability w
  u <- runif(N)  # uniform random samples
  ## Inverse CDF sampling of x
  x <- ifelse(m[J] != 0, (1.0/m[J])*log((m[J]*w[J]/beta[J])*u + exp(m[J]*z[J])), z[J] + w[J]*u/beta[J])

  return (list(x = x, J = J))
}
```

For the main loop we'll sample x in chunks that will grow until either the full number of samples is reached or until the chunk size equals N.

```
##### MAIN LOOP #####

## Loop until we have N samples (or until error)
while (length(x) < N) {

  ## Track iterations, 10000 is arbitrary upper bound
  current_iter <- current_iter + 1

  if (current_iter > max_iters) {
    stop('Maximum number of iterations reached.')
  }

  ## Vectorized sampling in chunks
  ## Chunk size will grow as square of iteration,
  ## up until it is the size of the sample (N)
  ## This ensures small samples at first, while
  ## u(x) is a poor approximation of h(x), and
  ## large samples when the approximation improves.
  chunk_size <- min(c(N, current_iter**2))


  ##### INITIALIZATION AND UPDATING STEP #####

  ## only re-initialize if there are new samples
  if (length(x0) > 0) {
```

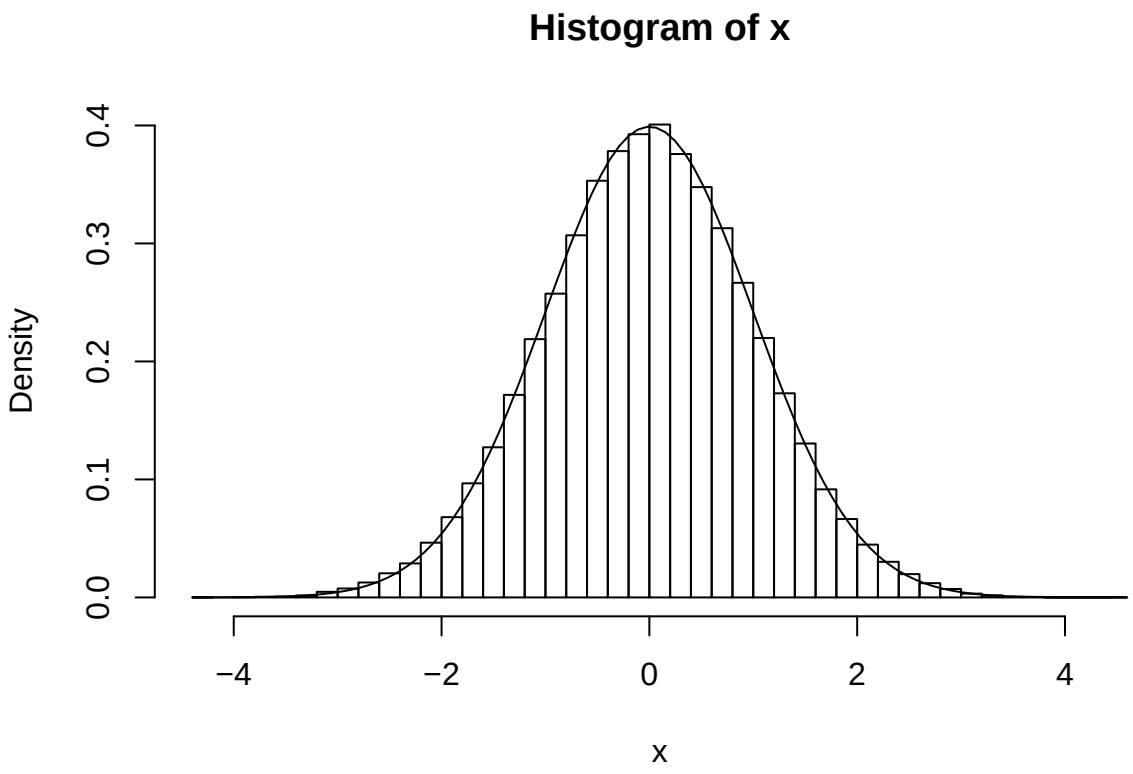
```
  ## Update with new values
  x_j <- append(x_j, x0)
  h_j <- append(h_j, h0)
  dh_j <- append(dh_j, dh0)
  x0 <- c()
  h0 <- c()
  dh0 <- c()

  ## Sort so that x_j's are in ascending order
  srtd <- sort(x_j, index.return = TRUE)
  x_j <- srtd$x
  h_j <- h_j[srtd$ix]
  dh_j <- dh_j[srtd$ix]

  L <- length(dh_j)

  ## Check for duplicates in x and h'(x)
  ## This prevents discontinuities when
  ## computing z_j's
  if (L > 1) {
    while (!all((abs(dh_j[1:L-1] - dh_j[2:L]) > eps) & ((x_j[2:L] - x_j[1:L-1]) > dx))) {

      ## Only keep values with dissimilar neighbors
      ## Always keep first index (one is always unique)
      dup <- append(TRUE, ((abs(dh_j[1:L-1] - dh_j[2:L]) > eps) & ((x_j[2:L] - x_j[1:L-1]) > dx)))
      x_j <- x_j[dup]
      h_j <- h_j[dup]
      dh_j <- dh_j[dup]

      L <- length(dh_j)
      if (L == 1) {
        break
      }
    }

    if (L > 1) {
      ## Ensure log-concavity of function
      if(!all((dh_j[1:L-1] - dh_j[2:L]) >= eps)) {
        stop('Input function f not log-concave.')
      }
    }
  }

  ## pre-compute z_j, u_j(x), l_j(x), s_j(x)
  z_j <- calc_z_j(x_j, h_j, dh_j, bounds)

  u_j <- calc_u(x_j, h_j, dh_j)
  m_u <- u_j$m
  b_u <- u_j$b

  l_j <- calc_l(x_j, h_j)
  m_l <- l_j$m
  b_l <- l_j$b
```

```
  s_j <- calc_s_j(m_u, b_u, z_j)
  beta_j <- s_j$beta
  w_j <- s_j$w

}

##### SAMPLING STEP #####

## draw x from exp(u(x))
draws <- draw_x(chunk_size, beta_j, m_u, w_j, z_j)
x_s <- draws$x
J <- draws$J

## random uniform for rejection sampling
w <- runif(chunk_size)

## Warn if samples were outside of bounds
## This happens if bounds given don't reflect
## the actual bounds of the distribution
if (!all((x_s > bounds[1]) & (x_s < bounds[2]))) {

  ## Flag so warning only happens once
  if (!bds_warn) {
    warning('Sampled x* not inside bounds...please check bounds.')
    bds_warn <- TRUE
  }

  ## Only keep x values within bounds
  ibd <- ((x_s > bounds[1]) & (x_s < bounds[2]))
  J <- J[ibd]
  x_s <- x_s[ibd]
  w <- w[ibd]
}

## Index shift for l_j(x)
J_l <- J - ifelse(x_s < x_j[J], 1, 0)
## only use x-values where l_j(x) > -Inf
ibd <- (J_l >= 1) & (J_l < length(x_j))

## Perform first rejection test on u(x)/l(x)
y <- exp(x_s[ibd]*(m_l[J_l[ibd]] - m_u[J[ibd]]) + b_l[J_l[ibd]] - b_u[J[ibd]])
acc <- (w[ibd] <= y)

## Append accepted values to x
x <- append(x, x_s[ibd][acc])
## Append rejected values to x0
x0 <- append(x_s[!ibd], x_s[ibd][!acc])

if (length(x0) > 0) {

  ## Evaluate h(x0)
  h0 <- h(x0)
  ## Finite difference approximation to h'(x)
```

```
    dh0 <- (h(x0 + dx) - h0)/dx

    ## Check for NaNs and infinities
    isnum <- is.finite(h0)&is.finite(dh0)
    x0 <- x0[isnum]
    h0 <- h0[isnum]
    dh0 <- dh0[isnum]

    w <- append(w[!ibd], w[ibd][!acc])[isnum]
    J <- append(J[!ibd], J[ibd][!acc])[isnum]

    ## Perform second rejection test on u(x)/h(x)
    acc <- (w <= exp(h0 - x0*m_u[J] - b_u[J]))
    ## Append accepted values to x
    x <- append(x, x0[acc])
  }
}

## Only return N samples (vectorized operations makes x sometimes larger)
x <- x[1:N]
```

Test the output.

```
hist(x, breaks = 50, freq = FALSE)
xr <- seq(min(x), max(x), length = 100)
lines(xr, dnorm(xr))
```



We see that the number of function evaluations goes approximately as $3n^{1/3}$.

```
print(length(x_j))
```

```
## [1] 131
```

```
print(3*N**(1/3))
```

```
## [1] 139.2477
```

This is consistent with the description of the efficiency of the method from Wilks.

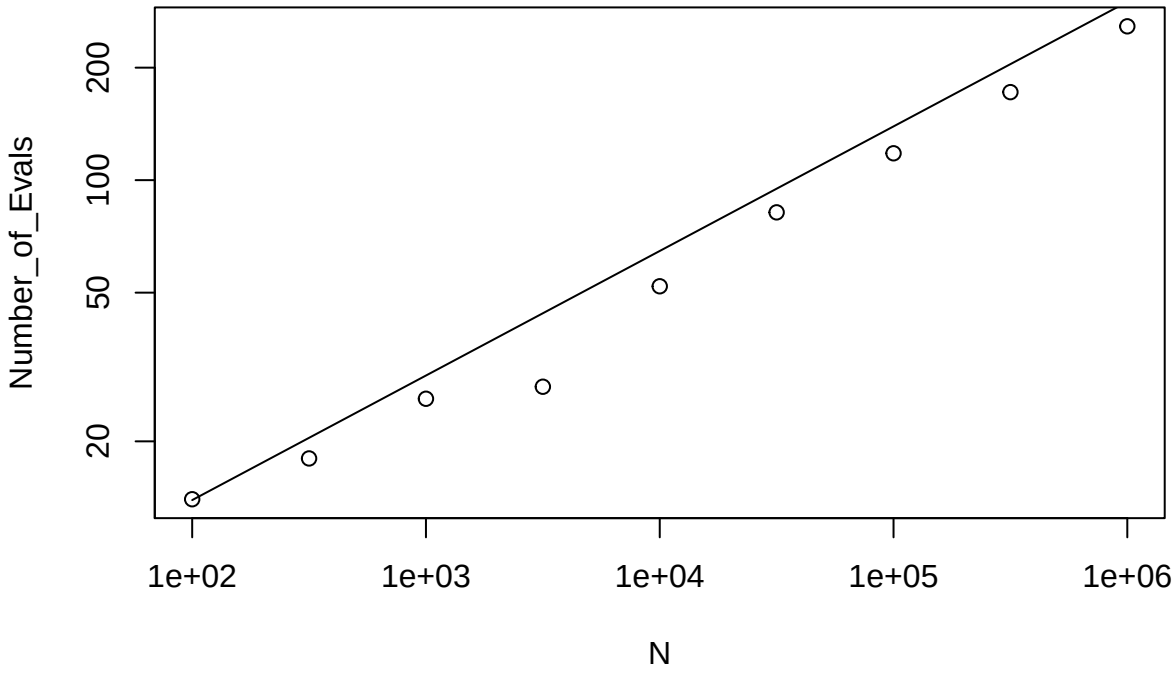
Plotting this for many sample sizes shows us the general trend.

```
library('ars')
```

```
##
## Attaching package: 'ars'
```

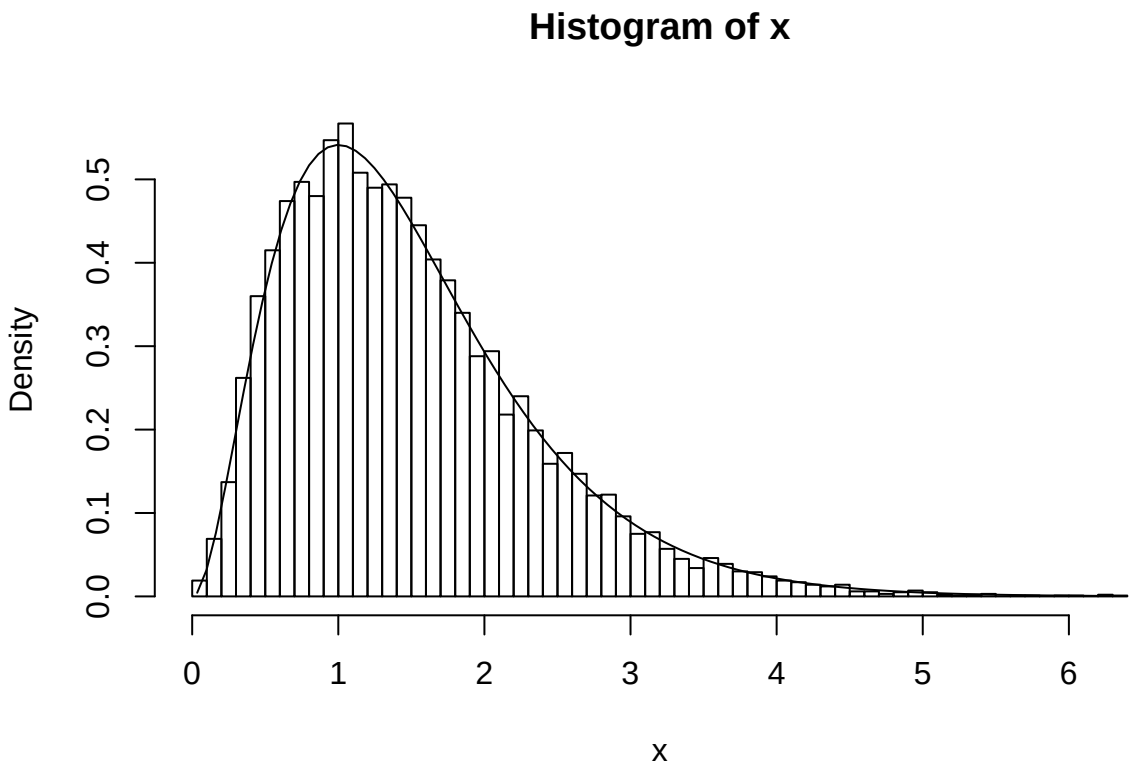
```
## The following object is masked _by_ '.GlobalEnv':
##
##     x
```

```
N_evals <- 0
f <- function (x) {
  N_evals <<- N_evals + 1
  return (dnorm(x))
}

Number_of_Evals <- c()
N <- 10**seq(2, 6, 0.5)

for (n in N) {
  x <- ars(f, n)
  Number_of_Evals <- append(Number_of_Evals, N_evals)
  N_evals <- 0
}

plot(N, Number_of_Evals, log = 'xy')
lines(N, 3*N**(1/3))
```



Example Usage

```
x <- ars(dgamma, 10000, x0 = c(0.1, 2.5), bounds = c(0.0, Inf), shape = 3, rate = 2)
hist(x, breaks = 50, freq = FALSE)
xr <- seq(min(x), max(x), length = 100)
lines(xr, dgamma(xr, shape = 3, rate = 2))
```



Testing