Adaptive Rejection Sampling

Jonathan Morrell, Ziyang Zhou, Vincent Zijlmans

December 12, 2018

General Methodology

Code

Helper Functions

```
f <- dnorm
N <- 100000
x0 <- c(-1.0, 1.0)
bounds <- c(-Inf, Inf)

## 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.')
}



## 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)
max_iters <- 10000  # prevent infinite loop
current_iter <- 0
bds_warn <- FALSE  # Flag for boundary warning in main while loop

## 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.')
}



##### HELPER FUNCTIONS #####

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]))

  ## 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)))
}

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))
}
```

```r
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))
}




##### 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
  while (!all(((dh_j[1:L-1] - dh_j[2:L]) > dx) & ((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, (((dh_j[1:L-1] - dh_j[2:L]) > dx) & ((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
    }
  }

  ## Ensure log-concavity of function
  if(!all(dh_j[2:L] <= dh_j[1:L-1])) {
    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
```

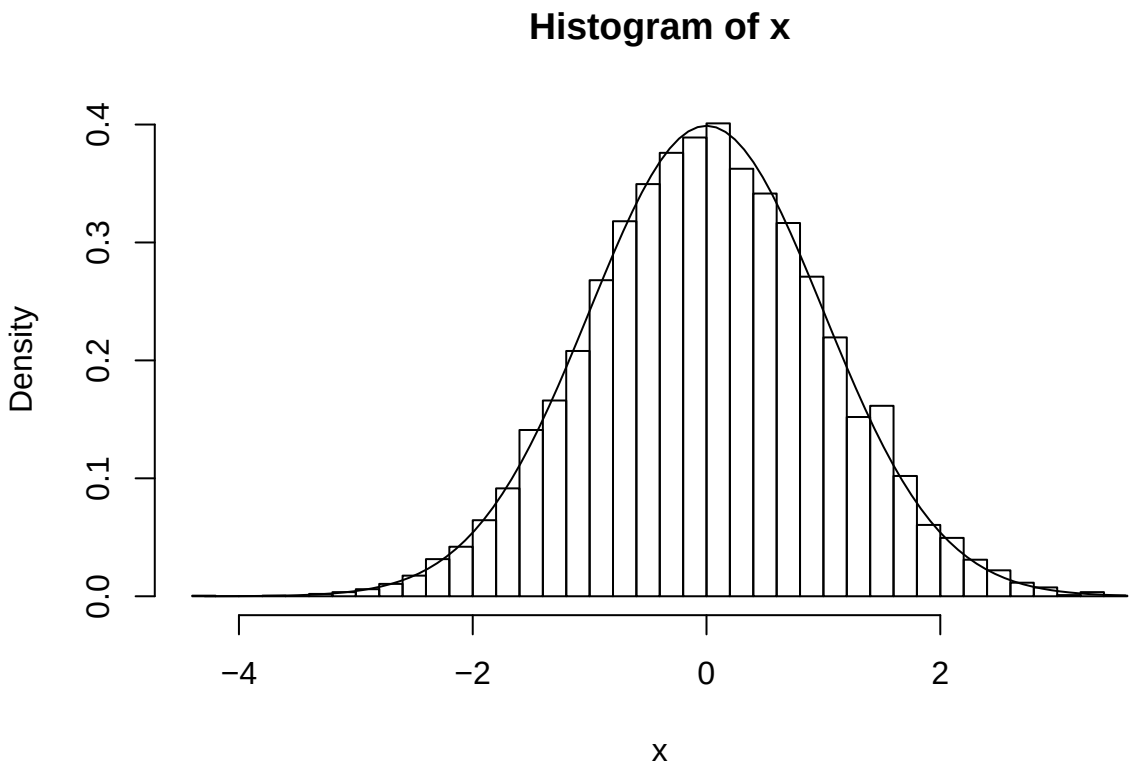
```
}

##### 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]

x[1:10]
```

```
##  [1] -0.98599079 -0.67278186 -0.09015947  2.00305499  0.11697261
##  [6] -0.83346206  0.83822732  0.78649207  0.70372606  0.68904207
```

Example Usage

```
library('ars')
x <- ars(dnorm, 10000)
hist(x, breaks = 50, freq = FALSE)
xr <- seq(min(x), max(x), length = 100)
lines(xr, dnorm(xr))
```




```
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))
```