# Adaptive Rejection Sampling

*Jue Wang, Li Yang, Winnie Gao*

*12/08/2018*

**Email Address:**

jue_wang@berkeley.edu, liyangc@berkeley.edu, zhaochen_gao@berkeley.edu

**Github Repository: WinnieGao1223/**

## 1. Approach

### (a) Functions

Modular Functions:

- 1. Deriv_sub: Take the derivative of a function evaluated at x numerically. Basic logic: If x is at the boundary:

$$f\prime(x) \approx \frac{f(x) - f(x-\epsilon)}{\epsilon}$$

$$f\prime(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

  Otherwise:

$$f\prime(x) \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

- 2. check_concave: Check the log-concavity of the density functions. Check if the derivative of function decreases monotonically with increasing x.
- 3. init_mat: We create a matrix called mat to store the information that's collected during the rejection sampling process. The first column is the value of sample x. The second column is function h (which is log of original f function) evaluated at x. The third column is the derivative of function h evaluated at x. This function initializes mat, fills it with starting points provided by the user or starting points we find if they put -Inf or Inf.

```
init_mat <- function(a,b,fun,deriv){
  #this mat stores x,f(x) and df(x)
  mat <- matrix(nrow = 2,ncol = 3)
  mat[1,1] <- a
  mat[1,2] <- fun(a)
  if(!is.finite(mat[1,2])){
    stop('a is not defined on function h', call. = FALSE)
  }
  mat[1,3] <- deriv(a,fun,a,b)
  mat[2,1] <- b
  mat[2,2] <- fun(b)
  if(!is.finite(mat[2,2])){
    stop('b is not defined on function h', call. = FALSE)
  }
  mat[2,3] <- deriv(b,fun,a,b)
```

```
  return(mat)
}
```

- 4. update_mat: Update mat with x_star, which is the proposed sample points generated by sampling process.

```
update_mat <- function(mat,x_star,fun,deriv,a,b){
  index <- sum(mat[,1] < x_star) #find the location to insert x*
  tmp <- matrix(NA,1,3)
  mat <- rbind(mat,tmp)
  mat[(index+2):nrow(mat),] <- mat[(index+1):(nrow(mat)-1),]
  mat[(index+1),] <- c(x_star,fun(x_star),deriv(x_star,fun,a,b))
  return(mat)
}
```

- 5. init_z: Initialize z, which is a vector used to store calculated z value.

```
init_z <- function(mat){
  z <- c()
  z[1] <- mat[1,1]
  z[3] <- mat[2,1]
  #if the slopes at a and b are equal, set z as the mean of a and b
  if (abs(mat[2,3]-mat[1,3]) < 1e-9){
    z[2] <- (mat[2,1] + mat[1,1])/2
  } else {
    z[2] <- calc_z(mat)
  }
  return(z)
}
```

- 6. update_z: Update z vector with new z values.

```
update_z <- function(z,x_star,mat){
  if (nrow(mat) <= 2){
    stop("something wrong with info matrix", call. = FALSE)
  }
  index <- which(mat[,1] == x_star)
  z <- c(z,NA)
  tmp_mat1 <- mat[(index-1):index,]
  tmp_mat2 <- mat[index:(index+1),]
  if (sum(abs(diff(mat[(index-1):(index+1),3]))) < 1e-9){
    z1 <- mat[(index-1),1] + (mat[(index+1),1] - mat[(index-1),1])/3
    z2 <- mat[(index-1),1] + (mat[(index+1),1] - mat[(index-1),1])/3*2
  } else {
    z1 <- calc_z(tmp_mat1)
    z2 <- calc_z(tmp_mat2)
  }
  if((round(z1, digits = 6) > round(z2, digits = 6))){
    stop('something wrong with updated z value, check log-convexity!',
         call. = FALSE)
  }
  z[(index+2):length(z)] <- z[(index+1):(length(z)-1)]
  z[index] <- z1
  z[index+1] <- z2
  return(z)
}
```

- 7. sample_x: Generate a new proposed sample point. We use information from mat to calculate $u_k(x)$ and $s_k(x)$ and then draw a random number from uniform distribution to determine the range of z we want to sample from. Then we draw another random number from uniform distribution to get the sample x through evaluating the inverse cdf at the value of random draw. The function would return the sample point.

```
sample_x <- function(z, mat, n=1){
  uk1 <- exp(mat[,2]+(z[-1]-mat[,1])*mat[,3])
  uk2 <- exp(mat[,2]+(z[-length(z)]-mat[,1])*mat[,3])
  p <- rep(NA,nrow(mat))
  p <- (uk1-uk2)/mat[,3]

  # normalize p
  p_norm <- p/sum(p)
  w <- runif(n)
  # determine the range we want to sample from
  i <- sum(cumsum(p_norm) < w) + 1

  # sample x using inverse cdf
  samp_x <- (log(p[i]*mat[i,3]*runif(n)+
                 exp(mat[i,2]+(z[i]-mat[i,1])*mat[i,3]))-mat[i,2])/mat[i,3]+mat[i,1]
  return(samp_x)
}
```

- 8. upper_bound: Calculate the upper bound. We take the z, mat and the newly generated sample x as input. We compare sample x with z to find which segment of line to calculate from, which is the index of smallest z that is larger than sample x. Then we use the linear relationship to get the upper bound.

$$u_k(x) = h(x_j) + (x - x_j)h\prime(x_j)$$

- 9. lower_bound: Calculate the lower bound. We take the mat and newly generated sample x as input. We compare sample x with first column of mat to find the index of largest x that is smaller than sample x and use the following formula to get the lower bound.

$$l_k(x) = \frac{(x_{j+1} - x)h(x_j) + (x - x_j)h(x_{j+1})}{x_{j+1} - x_j}$$

Main Part (ars): We try to find the appropriate starting point based on the information user provided and the log-concavity of the density function. We initialize mat and z vector. While the sample size does not reach our desired number, we repeat the rejection sampling process. First, we check the log-concavity of the density function every time we update the mat. Then we generate one proposed sample x using sample_x function. We draw one random number (w) from the uniform distribution and also calculate the upper bound (u) and lower bound (l) using corresponding functions. Then we determine if:

1. w <= exp(l-u), then accept the sample x and store it in the final result vector.
2. w <= exp(h(sample x)-u), then accept the sample x, store it in the final result vector, update the mat and z with this sample x.
3. Otherwise, reject the sample x and update the mat and z with this proposed sample x.

## (b) Vectorization

We use matrix to store the updated information and store z and output as vectors. # 2. Testing ## (a) Input Checking We wrote various tests to check if the inputs are valid, including check if the class of input function is actually a function, check if the starting points and sample size are numbers. We also checked

whether the input function is log-concave and whther the function is defined on the given value. ## (b) Unit Tests

### (c) Formal Testing

## 3. Contributions

Li, Jue and Winnie worked together to create the first draft of all the modular functions, and the main body of ars function. Then Jue started working on building the test cases for modular functions. And Li contributed to the test cases included in the ars function. Winnie wrote the codes about how to find the appropriate starting points where f(x) is defined. Then Jue took care of coding about special cases like uniform distribution case. Li wrote the formal testing. Winnie wrote the overall help page and compiled the first draft of writeup. All of three members contributed to revisions. We worked together to make everything into a package.