Adaptive Rejection Sampling

Katherine Kempfert, Eric Chu, Jiahui Zhao

December 18, 2020

Introduction

Adaptive rejection sampling is a method for efficiently sampling from univariate probability density function which is log-concave. It is particularly useful when the distribution interested is computationally expensive. In this project, our group implemented the Adaptive Rejection Sampler (ars) based on algorithms discussed in the paper *Adaptive Rejection Sampling for Gibbs Sampling* by W. R. GILKS. More details about the algorithm is shown in Methods section.

Methods

To make sure tangent lines can be used as upper bounds, our function `ars` assumes the input density is log concave, that is, $h(x) = log(g(x))$ is a concave function. After initializing valid x abscissas, we calculate the intersections of tangent lines using

$$z_j = \frac{h(x_{j+1}) - h(x_j) - x_{j+1}h'(x_{j+1}) + x_j h'(x_j)}{h'(x_j) - h'(x_{j+1})}$$

For $z \in [z_{j-1}, z_j]$ and $j = 1, 2, ..., k$, the upper bound is defined and calculated as

$$u_k(x) = h(x_j) + (x - x_j)h'(x_j)$$

The sampling density $s_k(x)$, which we will use to draw samples from is

$$s_k(x) = \frac{\exp u_k(x)}{\int_D \exp u_k(x')dx'}$$

Observations will be sampled as follows. First, we find the interval to which x will be sampled from by selecting one of the piece of the piece-wise exponential density curves, which have been normalized using the denominator in above function $s_k(x)$. Then we randomly generate a value u_1 from $Uniform(0, 1)$ distribution, and find the largest interval index, i, such that the total integral value from the lowest interval of x to the upper bound of that interval is smaller than u_1.

We then use the Inverse CDF method to actually draw x^* within the i_th interval.

The CDF, given that x belongs to a particular interval, is computed as

$$S(x) = P(X \leq x | x \in [z_{j-1}, z_j]) = \frac{\int_{z_{j-1}}^{x} \exp u_k(x')dx'}{\int_{z_{j-1}}^{z_j} \exp u_k(x')dx'}$$

$S(x)$ is a value between 0 and 1. The denomination of $S(x)$ is a normalizing constant and can be denoted as C.

$$C = \int_{z_{j-1}}^{z_j} \exp u_k(x')dx'$$

The numerator can be integrated as follows,

$$\begin{aligned}
\int_{z_{j-1}}^{x} \exp u_k(z)dz &= \int_{z_{j-1}}^{x} \exp(h(x_j) + (z - x_j)h'(x_j))dz \\
&= \exp(h(x_j) - x_j h'(x_j)) \int_{z_{j-1}}^{x} \exp(zh'(x_j))dz \\
&= \exp(h(x_j) - x_j h'(x_j)) \cdot \frac{\exp(zh'(x_j))}{h'(x_j)}\Big|_{z_{j-1}}^{x} \\
&= \frac{\exp(h(x_j) - x_j h'(x_j))}{h'(x_j)} \left(\exp(xh'(x_j)) - \exp(z_{j-1}h'(x_j))\right).
\end{aligned}$$

Then, we randomly generate a value from $Uniform(0,1)$ distribution, u_2.

$$\frac{\int_{z_{j-1}}^{x^*} \exp u_k(z)dz}{C} = u_2$$

and,

$$\int_{z_{j-1}}^{x^*} \exp u_k(z)dz = u_2 \times C$$

The inverse CDF can be computed as

$$\frac{e^{h(x_j)-x_j h'(x_j)}}{h'(x_j)}(e^{x^* h'(x_j)} - e^{z_{j-1}h'(x_j)}) = u_2 \times C$$

$$x^* = \frac{1}{h'(x_j)} log(\frac{u_2 \times C \times h'(x_j)}{e^{h(x_j)-x_j h'(x_j)}} + e^{z_{j-1}h'(x_j)})$$

After getting a new sample x^*, we perform the squeezing test and rejection test with a randomly generated uniform value w.

We accept x^* when either $w \leq e^{l_k(x^*)-u_k(x^*)}$ (squeezing test) or $w \leq e^{h(x^*)-u_k(x^*)}$ (rejection test) is met. However, we only include x^* in T_k to form T_{k+1} when the rejection test is evaluated.

We repeat the above algorithm until we get enough samples.

Implementation

The main function is `ars()`, which takes four arguments:

- g: the unvariate function from which to sample; log(g) must be a log concave function
- bounds: a length 2 vector defining the lower and upper bounds for the distribution g
- n: the number of observations to be sampled

- initial: the initial value of x abscissae which can be defined by users

```r
ars(g = dnorm, bounds = c(-Inf, Inf), n = 1000, initial = NULL)
```

The output of main function is a length-n vector of samples generated from distribution g.

To make the code modular and portable, we created different helper functions and organized them into various files. Each file only contain auxiliary functions related a specific calculation need. Important auxiliary functions are listed below.

- In file `lk.R`
  - `calc_lkj` takes two abscissa values x_j, x_{j+1}, log concave function h as inputs. It defines and returns a function to calculate the value of l_k at a specific x point.
  - `calc_lk` takes log concave function h and the abscissa vector T_k as inputs. It defines and returns a list of l_k functions.
  - `get_lk_x` takes a x value, abscissa vector T_k and the list of l_k functions returned by `cal_lk` function as inputs. It calculates and returns the value of l_k at a specific point x.
- In file `sk.R`
  - `integrate_exp` takes values for lower bound, upper bound, log concave function h and one abscissa x_j and its derivative as inputs. It takes integral over the exponential u_k function. It returns a integral value from lower boundary to upper boundary.
  - `sample_sk` takes the abscissa vector, log concave function h and the list of u_k functions as inputs. It generates and returns a sample from an interval of s_k.
- In file `uk.R`
  - `calc_uk` takes log concave function h and abscissa vector T_k as inputs. It defines and returns a list of u_k function for all x in T_k.
  - `get_uk_x` takes a x value, intersection vector z returned by `calc_z` function and list of u_k function return by `calc_uk` function as inputs. It calculates and returns the value of u_k at a specific point x.
- In file `zk.R`
  - `calc_zj` takes two abscissa values x_j, x_{j+1} and the log concave function `h` as inputs. It defines the function to calculate the intersection of tangent line. It returns a function of z_k as output.
  - `calc_z` takes the boundary vector, abscissa vector T_k and log concave function h as inputs. It calculates intersections of consecutively paired tangent line. The output is a vector of intersections, z, bounded by two elements in vector boundary.
- In file `ars.R`
  - `calc_init_vals` takes the distribution g and the numeric vector bounds as inputs. It initializes and returns abscissa x values by considering four cases: interval D has both lower and upper bounds, only has upper bound, only has lower bound and interval D is unbounded. This function returns a vector of x abscissa.

Result

In this section, we will see how our **ars** function performs using graphs.

```
# install.packages("ars_0.1.0.tar.gz", repos = NULL, type = "source")
library(ars)
```

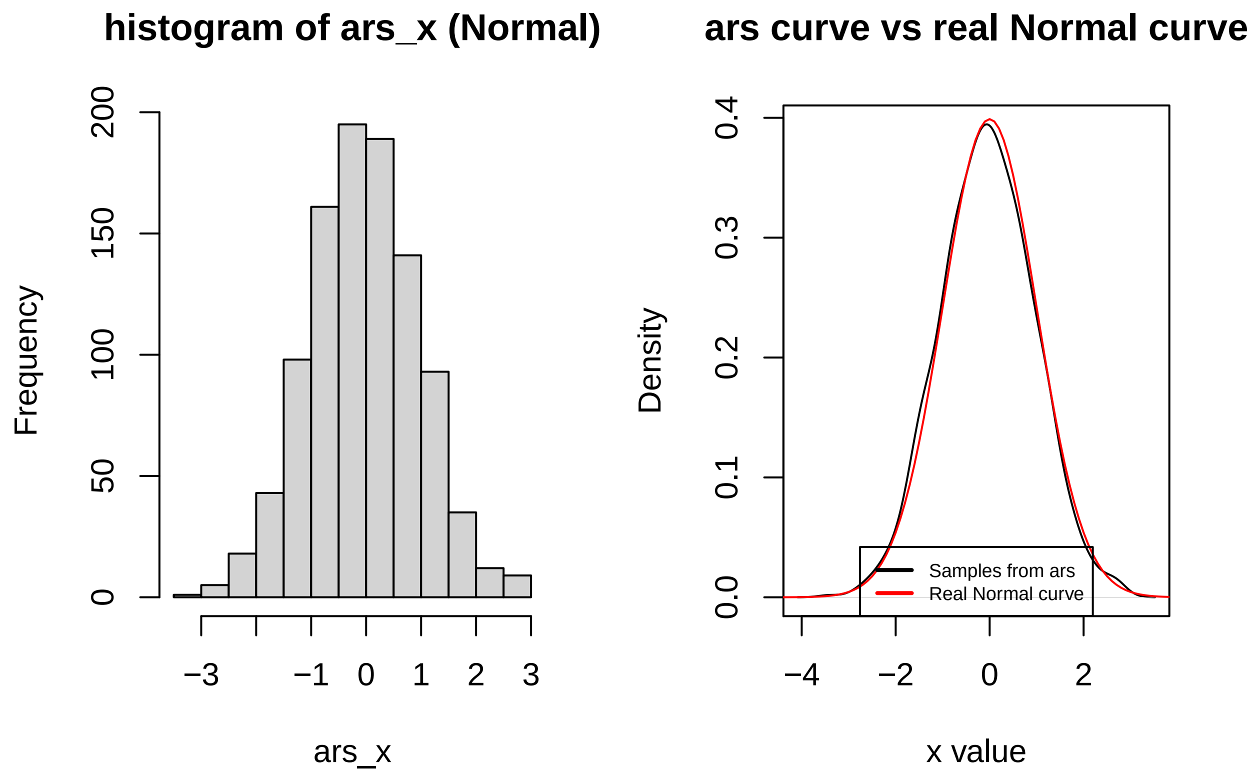
Standard normal distribution

```
set.seed(24)
ars_x <- ars(g=dnorm, n=1000, initial = NULL, bounds = c(-Inf, Inf))
```

```
par(mfrow = c(1, 2))
x <- seq(-5, 5, by = 0.1)

hist(ars_x, main = "histogram of ars_x (Normal)")

plot(density(ars_x), xlab = "x value", main = "ars curve vs real Normal curve")
lines(x, dnorm(x), col = 'red')
legend("bottom", c("Samples from ars", "Real Normal curve"),
       col = c("black", "red"),
       lwd = 2,
       cex = 0.6)
```



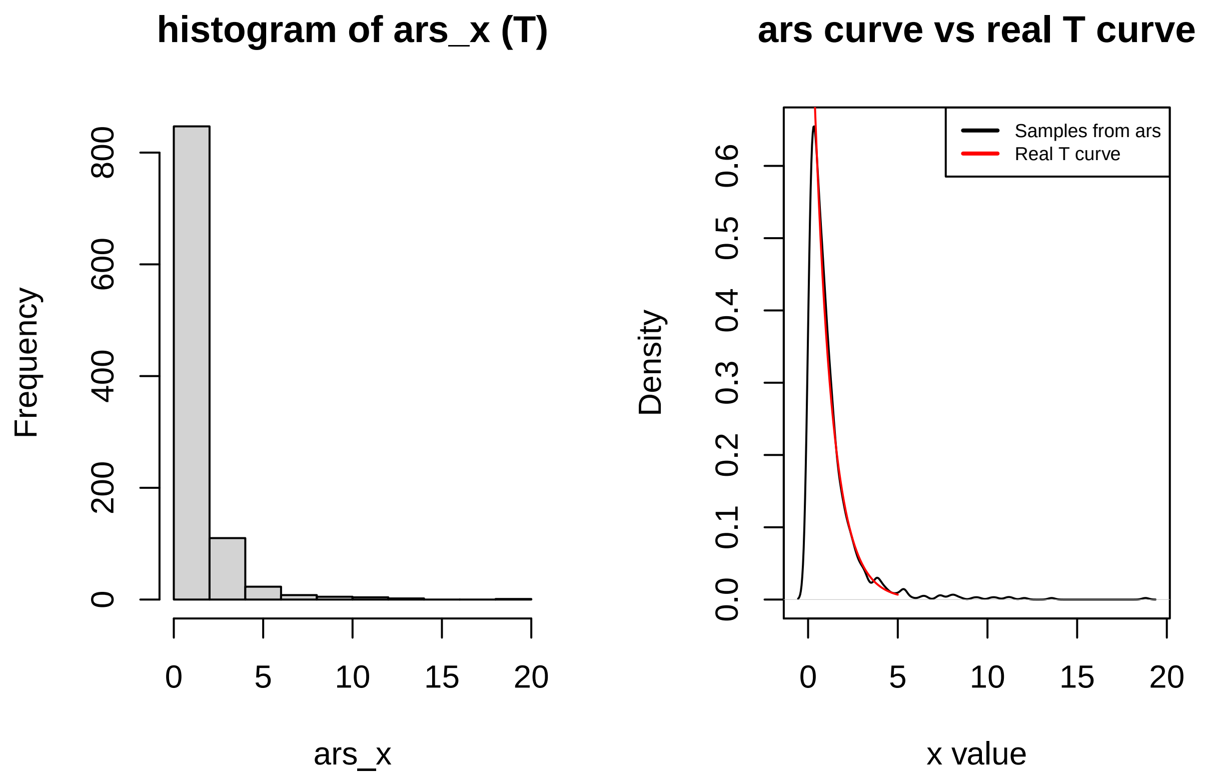
Truncated T distribution with df = 2

```r
set.seed(24)
t_dist <- function(x) {dt(x, df = 2)}
ars_x <- ars(g=t_dist, n=1000, initial = NULL, bounds = c(0, Inf))
```

```r
par(mfrow = c(1, 2))
x <- seq(0, 5, by = 0.1)

hist(ars_x, main = "histogram of ars_x (T)")

plot(density(ars_x), xlab = "x value", main = "ars curve vs real T curve")
lines(x, dexp(x), col = 'red')
legend("topright", c("Samples from ars", "Real T curve"),
       col = c("black", "red"),
       lwd = 2,
       cex = 0.6)
```



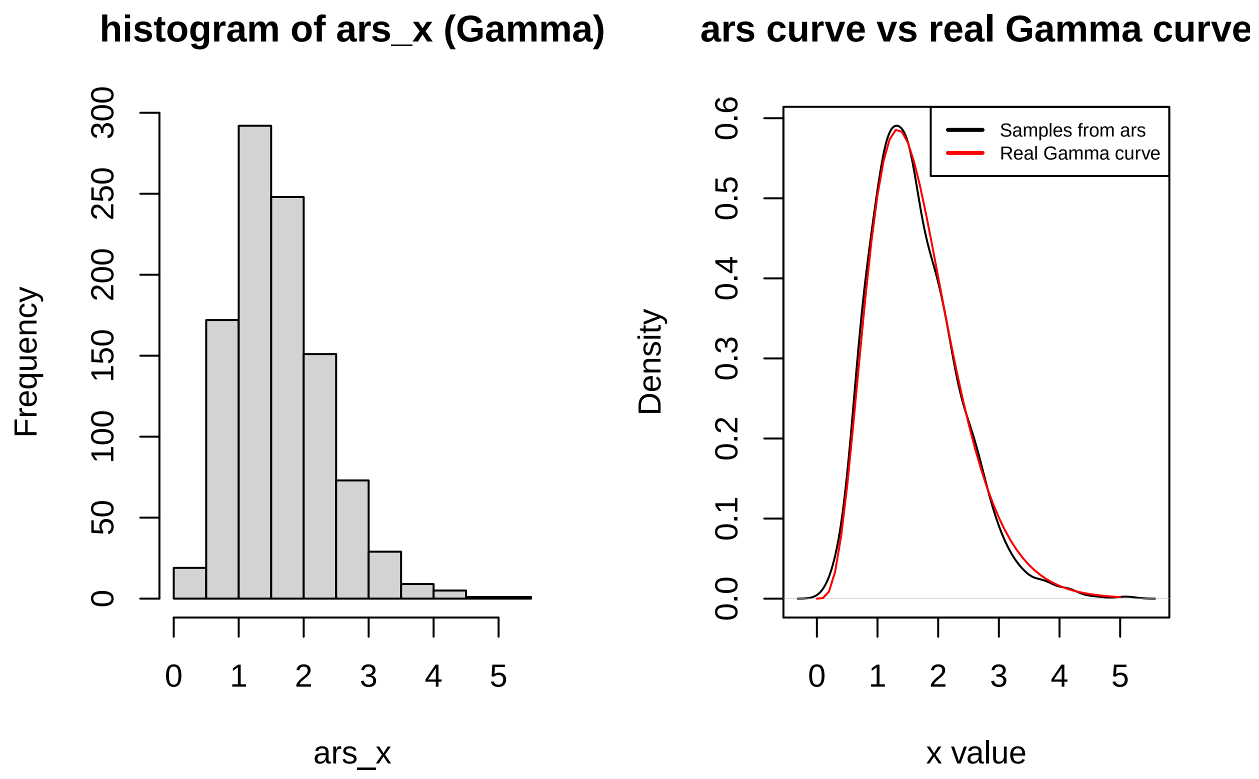
Gamma Distribution with shape = 5 and rate = 3

```r
set.seed(24)
gamma_dist <- function(x){dgamma(x, shape = 5, rate=3)}
ars_x = ars(g=gamma_dist, n=1000, initial = NULL, bounds = c(0, Inf))
```

```r
par(mfrow = c(1, 2))
x <- seq(0, 5, by = 0.1)

hist(ars_x, main = "histogram of ars_x (Gamma)")

plot(density(ars_x), xlab = "x value", main = "ars curve vs real Gamma curve")
lines(x, dgamma(x, shape = 5, rate = 3), col = 'red')
legend("topright", c("Samples from ars", "Real Gamma curve"),
       col = c("black", "red"),
       lwd = 2,
       cex = 0.6)
```



In all three examples above, we can see curves with samples generating using `ars` packages really close to the real density curves.

Test

```r
# devtools::test()
```