STAT 243 Final Project: ARS R Package

Jeffrey Kuo, Hsiang-Chuan Sha, Tessa Weiss

2022-12-14

Package Installation

The files for creating the `ars` package can be found at https://github.berkeley.edu/tweiss/ars.git. In order to install the package in RStudio, refer to the following instructions. Note that required packages are `assertthat`, `numDeriv`, `stats`, `rmutil`, and `testthat`. Also remember to clean your environment before installing the package to ensure there are no errors with installation.

1. clone the repository to your local environment using `git clone "https://github.berkeley.edu/tweiss/ars.git"`
2. set your working directory in RStudio to the `ars` directory you just cloned
3. type `devtools::load_all()` into your console
4. `library(ars)`

Introduction

This project aims to create an R package called `ars` that simulates the adaptive rejection sampling algorithm described in the paper Adaptive Rejection Sampling for Gibbs Sampling by W.R. Gilks and P Wild. In this paper, Gilks and Wild propose a method for rejection sampling, which is valid for any univariate, log-concave probability density function. As part of this project, we created a main file called `ars.R` which contained the main `ars()` function used to calculate and produce the samples, with helper functions to complete each step in the algorithm defined in `ars_functions.R`.

Approach

The main function to calculate and produce the samples is called `ars()`. For purposes of organization and readability, we decided to use a functional programming approach where we defined a number of auxiliary functions to complete subtasks of the `ars()` function, and wrote all of these functions in a separate R script entitled `ars_functions.R`. The main `ars()` function takes in the following inputs from the user:

- `f` is the density function from which the user wants to sample from
  - requirement of the argument is that it must be a univariate, log-concave function
- `n` is the number of samples the user wants to generate from the function `f`
  - requirement of the argument is that it must be a positive integer value
- `bounds` denotes the left and right bounds of the domain of the function `f`, which is the set of points x for which $f(x) > 0$
  - requirement of the argument is that it be a vector of length 2, where the first element is the left (lower) bound and the second element is the right (upper) bound
- `x_init` is the initial point within the bounds used as one of the initial abscissae points for future sampling
  - requirement of the argument is that it is a single point defined within the bounds

We note that if the user does not provide the argument `n`, the default number of samples is set to 1000, and if the user does not provide the argument `bounds`, the default bounds are set to (-Inf, Inf). If the user does not provide the argument `x_init`, we define `x_init` by optimizing the function `f` with the following code:

```
if (is.na(x_init)) {
    x_init <- optim(1, f, method = 'L-BFGS-B', control = list(fnscale=-1))$par
    }
```

The `control = list(fnscale=-1)` parameter allows us to find the value of the maximum of the function as opposed to the minimum, which is the default behavior of the `optim()` function. By finding the point x that maximizes the function f (i.e. the mode), we ensure that we can initialize the abscissae with points of positive probability.

The first task the function carries out is checking the validity of the user inputs. Thus, we implemented checks to ensure that `f` is a function, `n` and `x_init` are numeric, `bounds` is a vector of length two such that the first element is less than the second element (i.e. the lower bound is smaller than the upper bound), and that `x_init` is a point located within the bounds. After passing these initial argument checks, the `ars()` function moves on to defining two simple functions: `h()` and `hprime()`, where both functions take in a point x. `h()` returns the value $\log f(x)$ and `hprime()` returns the derivative of this value; that is, $\frac{d}{dx}\log f(x)$, where we used the external package `numDeriv` to compute the derivative. Next, the `ars()` function is broken down into the three steps Gilks and Wild suggest: (1) initialization, (2) sampling, and (3) updating.

Initialization Step

The basic idea of the initialization step is to initialize the abscissae points, which we define in the function as `Tk`. In order to carry out this step, we define an auxiliary function `initialize_abscissae()`, which uses the `x_init` point as a reference in order to output a vector of $k = 20$ initial abscissae points, which we then pass through the functions `h()` and `hprime()`. After defining these 20 initial abscissae points and computing their derivatives, we initialize the functions $u_k(x), s_k(x), l_k(x)$ and calculate the intersection of the tangent lines, z_j, for each x_j and x_{j+1}, where $j = 1, \ldots, k-1$. As described in the paper by Gilks and Wild, we derive the rejection envelope on T_k and calculate the value of every x_k on the rejection envelope, which was done utilizing the function $u_k(x)$. This function first identifies the segment where the inputted value x lies, and calculates the $u_k(x)$ for the corresponding slope and intercept of the piecewise function. We similarly calculate the squeezing function $l_k(x)$ with our function $l_k(x)$. Therefore, we derived both the rejection sampling value and the squeezing value for each point within T_k from $u_k(k)$, and $l_k(x)$, respectively.

For $s_k(x)$, we approximate the density function by calculating the area under the piecewise $e^{u_j(x)}$ for each segment (z_j, z_{j+1}), including the bounds the function is defined on as z_0 and z_k, in order to find the unnormalized probability q_j, which takes the following form:

$$q_j = \int_{z_j}^{z_j+1} e^{u_j(x)} dx$$

As defined in the paper, $u_j(x) = h(x_j) + (x - x_j)h'(x_j)$, which simplifies the integral to two cases based on the value of $h'(x_j)$:

$$q_j = \begin{cases} e^{u_j(z_j)}(z_{j+1} - z_j) & \text{if } h'(x_j) = 0 \\ \frac{e^{u_j(z_{j+1})} - e^{u_j(z_j)}}{h'(x_j)} & \text{if } h'(x_j) \neq 0 \end{cases}$$

The normalized probability of sampling each segment is derived by dividing each q_j with the summation of all of the unnormalized probabilities $\sum_{j=1}^{k+1} q_j$. The probability to sample each interval (z_j, z_{j+1}) is therefore equal to

$$p_j = \frac{q_j}{\sum_{j=1}^{k+1} q_j}$$

Sampling Step

We followed the sampling method that Gilks and Wild derived. That is, we first sample the segment using the probabilities calculated above. Then we sample an x^* with our `sample_sk()` function by applying the inverse CDF of that segment to a sampled draw of w from a $Unif(0,1)$ distribution which represents the probability that X is less than or equal to x^*:

$$w = P(X \leq x^*) = \frac{\int_{z_j}^{x^*} e^{u_j(x)} dx}{\int_{z_j}^{z_{j+1}} e^{u_j(x)} dx}$$

where the denominator is the area under the curve of the sampled interval calculated above. Let $\int_{z_j}^{z_{j+1}} e^{u_j(x)} dx = A_j$ for simplicity. For $h'(x_j) \neq 0$, we solve for x^* in the equation:

$$\begin{aligned} w &= \frac{\int_{z_j}^{x^*} e^{u_j(x)} dx}{A_j} \\ \implies x^* &= \begin{cases} \frac{\log\left(wAh'(x_j) + e^{u_j(z_j)} - h(x_j)\right)}{h'(x_j)} + x_j & \text{if } h'(x_j) \neq 0 \\ z_j + w(z_{j+1} - z_j) & \text{if } h'(x_j) = 0 \end{cases} \end{aligned}$$

where the $h'(x_j) = 0$ case is akin to scaling the uniform by the length of the interval.

We then perform the following tests:

First is the squeezing test: if $w \leq e^{l_k(x^*) - u_k(x^*)}$, we add the value x^* to our sample vector. If this is not the case, we move on to the rejection test: if $w \leq e^{h(x^*) - u_k(x^*)}$, we add the value x^* to our sample vector. Otherwise, x^* is rejected. In the event a value of x^* leads us to perform the rejection test, we also complete the updating step, defined in the next section.

Updating Step

As just mentioned, if x^* fails the squeezing test, we perform the updating step, which works as follows: we append the value of x^* to our vector T_k such that we now have T_{k+1}. We then sort the values of T_{k+1} so they are in ascending order, and compute $h'(x^*)$, placing it in its analogous location to its position in T_{k+1}. We then calculate the new intersection points of the tangent lines in of the points in T_{k+1} and return to the beginning of the sampling step. We complete this process until we have obtained a total of n samples, and the vector of n samples are returned from the `ars()` function to the user.

Notes on the Algorithm

While designing the code, one issue we came across was in comparing if two elements in a vector were equal. As a simple example, when trying to see if `x-y==0` for two elements `x` and `y` which were the same, we would sometimes see a result of `FALSE`. We figured out that this was because of a numerical precision error, and our solution to this was to define a tolerance for a margin of error, which we chose to be the square root of Machine Epsilon, or 1e-8, thus rewriting the above check as `abs(x-y)<=1e-8`, which would return `TRUE` as desired. Another issue we came across was in checking that the function `f` that the user provided was indeed a log-concave function by performing checks as our calculations proceeded rather than just checking if the entire function was log-concave at the beginning of our `ars()` function. The way we solved this was to use the definition of log-concavity noted in the Gilks and Wild paper. That is, a function $f(x)$ is log-concave if $h'(x)$ decreases monotonically with increasing x. Thus, we defined a function `check_log_concave()` which ensured this condition, using the same strategy to account for numerical precision as mentioned above. Below is the implementation:

```r
check_log_concave <- function(x) {
  n <- length(x)
  # only one element means log-concave
  if (n == 1) {
    return(0)
  }
  assertthat::assert_that(sum(x[2:n] - x[1:(n-1)] <= 1e-8) == n-1,
                          msg = "Function is not log-concave")
}
```

We ran this function every time we defined new points in T_k (and thus calculated a new $h'(x)$), and this seemed to work in ensuring that the function `f` provided by the user was indeed log-concave.

Tests

In order to run our specified set of tests, refer to the following instructions. Note that you may have to run the tests multiple times to get a 100% pass rate, as our tests rely on the p-value of the Kolmogorov-Smirnov test, so some tests may fail due to stochasticity.

1. In order to test the main `ars()` function: `testthat::test_file("tests/testthat/test-ars.R")`
2. In order to test the auxiliary functions: `testthat::test_file("tests/testthat/test-ars_functions.R")`

In order to run the tests for our function, we created an R script which utilized the R package `testthat`. We designed a number of tests in order to ensure that our function works as desired. The first set of tests was designed to ensure that the `ars()` function errors out when the user inputs a non-log-concave function `f`. Some known densities that are not log-concave include the pareto distribution, the t-distribution, and the lognormal distribution. Thus, we inputted these densities as our function `f` to make sure the `ars()` function returned an error. The second set of tests was designed to ensure that the `ars()` function errors out when the user enters invalid inputs. Some example tests we ran were inputting a value for `f` that was not a function and inputting a non-numeric argument for the number of samples `n`. The last set of tests was designed to check that our `ars()` function returned a vector of samples that resembled a vector of samples of the same length from its respective distribution. In order to do this, we implemented the Kolmogorov-Smirnov test. As a basic summary, the Kolmogorov-Smirnov test is a nonparametric test based on the empirical CDF function that tests the equality of samples from a continuous distribution with samples from a reference distribution, given that the distribution is fully specified (i.e. rate, shape, etc. parameters are provided). The Kolmogorov-Smirnov test statistic is defined as

$$D = \sup_x |F_0(x) - F_{data}(x)|,$$

where $F_0(x)$ is the CDF of the hypothesized distribution ("true" distribution) and $F_{data}(x)$ is the empirical CDF of the observed data (data coming from `ars()`). The null and alternative hypotheses are

$$H_0 : \text{the data comes from the specified distribution}$$
$$H_a : \text{at least one value does not match the specified distribution}$$

In order to run this test in R, we used the function `ks.test()` and ensured that the obtained p-value was greater than $\alpha = 0.05$. We ran the test on multiple known distributions in R (Normal, Exponential, Beta, Uniform, and Gamma), where we generated a vector of n samples for a known distribution using our `ars()` function, and compared it to a specified "true" distribution, which was a same-length vector of samples from the known distribution using built-in R functions, (`rnorm()`, for example).

We also wrote an R script to test that our auxiliary functions were providing reasonable results; namely `check_log_convave()`, `initialize_abscissae()`, `calc_z()`, `u()`, `l()`, `calc_probs()`, and `sample_sk()`. The tests for `check_log_convave()` aimed at targeting vectors with non-decreasing values, and we also included a test case for a vector where every value was the same (as this would still be log-concave). The tests for `initialize_abscissae()` aimed to try each case where the left bound was -Inf, the right bound was Inf, or both bounds were finite, ensuring that a vector of length $k = 20$ was properly returned for each test case. For testing `calc_z()`, `u()`, `l()`, `calc_probs()`, and `sample_sk()`, we provided sample values for the functions to perform computations on, and then compared the result of the function with a manual computation carrying out the same task for a variety of test cases. Given that each of these functions work under a "black-box" methodology, the sample values provided were random and contained no real meaning, but were still effective in testing the basic tasks of each function.

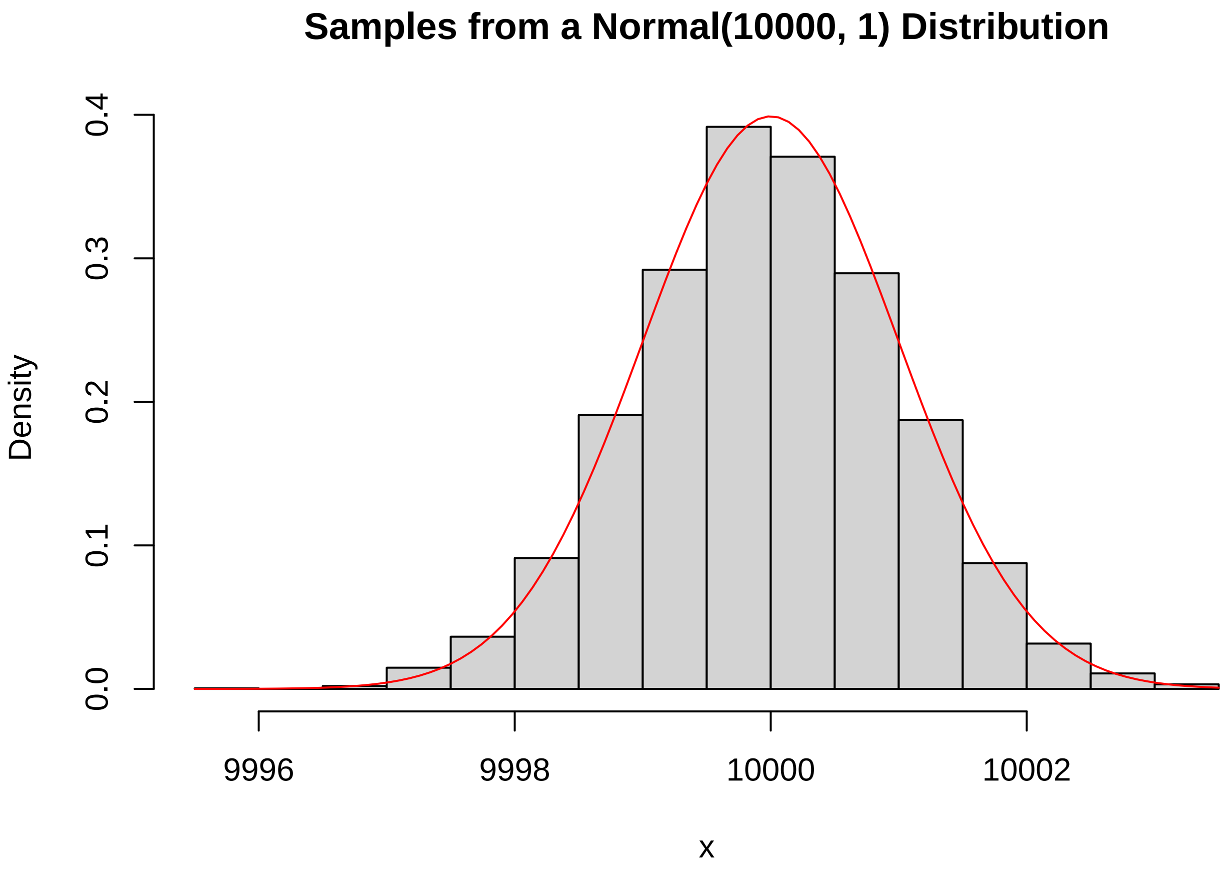
Examples

To visualize the results of our `ars()` function, below is an implementation of the function utilizing some common densities with a histogram of the samples compared to the actual curve of the distribution displayed on top.

```r
devtools::load_all()
library(ars)
```

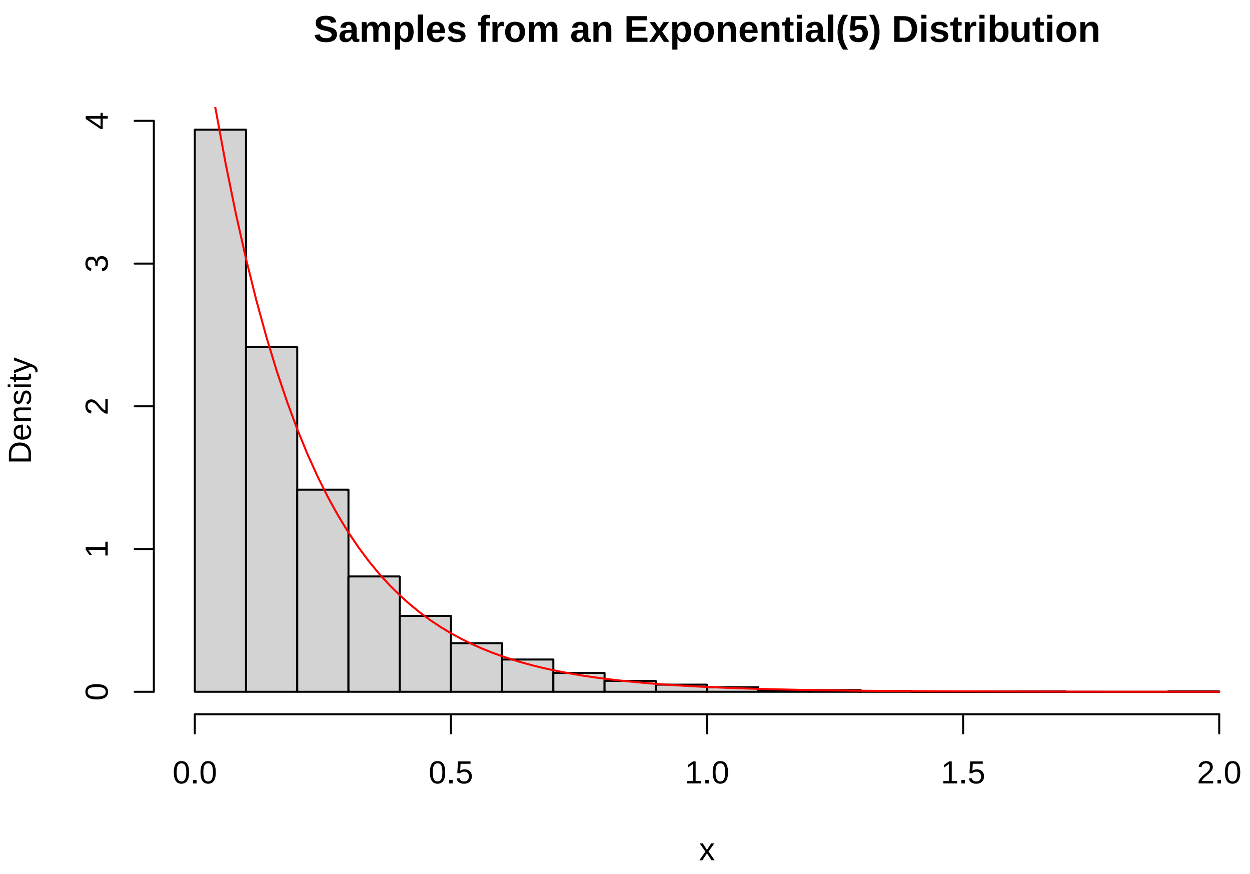
Example 1: Normal

```r
samps <- ars(function(x){dnorm(x, 10000, 1)}, n = 5000, bounds = c(-Inf, Inf), x_init = 9999)
hist(samps, xlab = "x", main = "Samples from a Normal(10000, 1) Distribution",
     freq = F)
curve(dnorm(x, 10000, 1), col = "red", add = T)
```



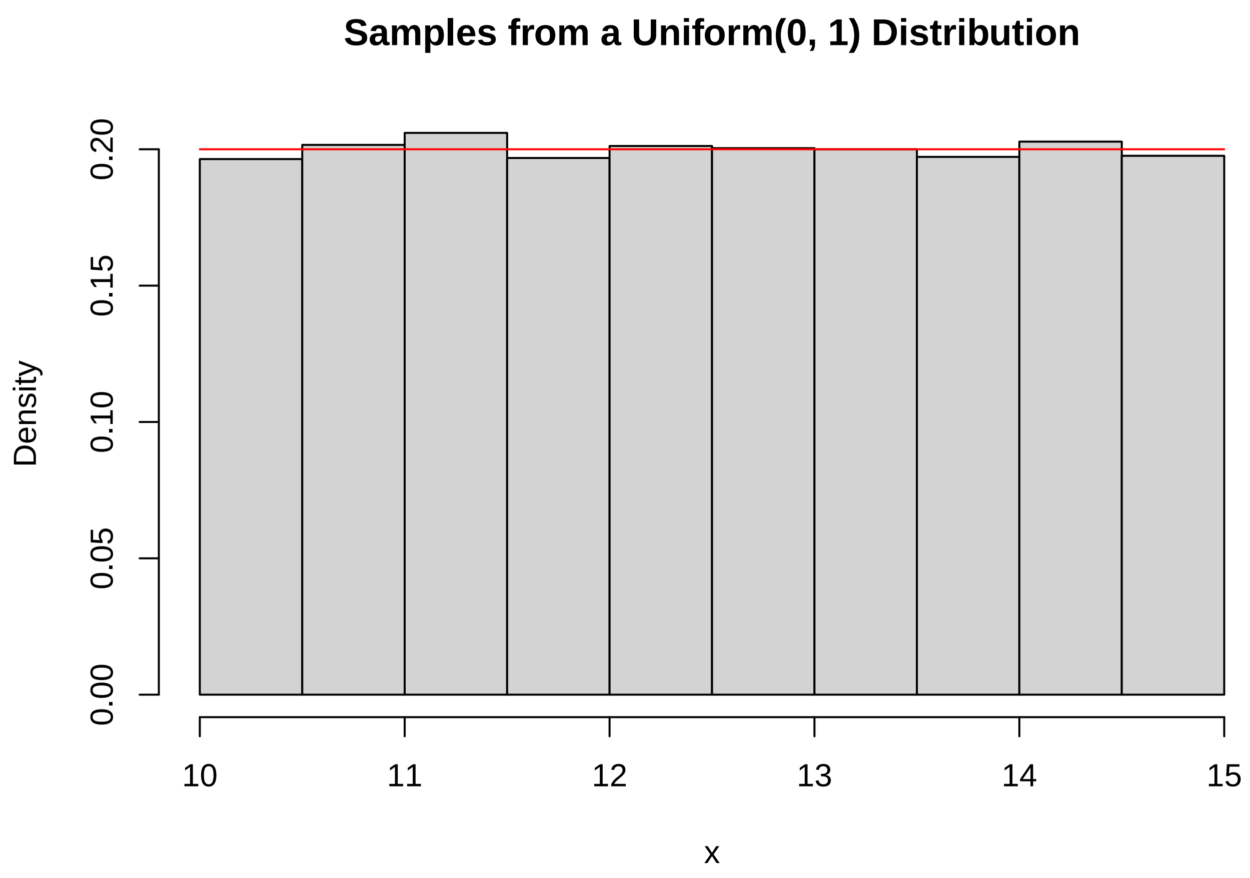
Example 2: Exponential

```
samps <- ars(function(x){dexp(x, 5)}, n = 5000, bounds = c(0, Inf), x_init = 1)
hist(samps, xlab = "x", main = "Samples from an Exponential(5) Distribution",
     freq = F)
curve(dexp(x, 5), col = "red", add = T)
```



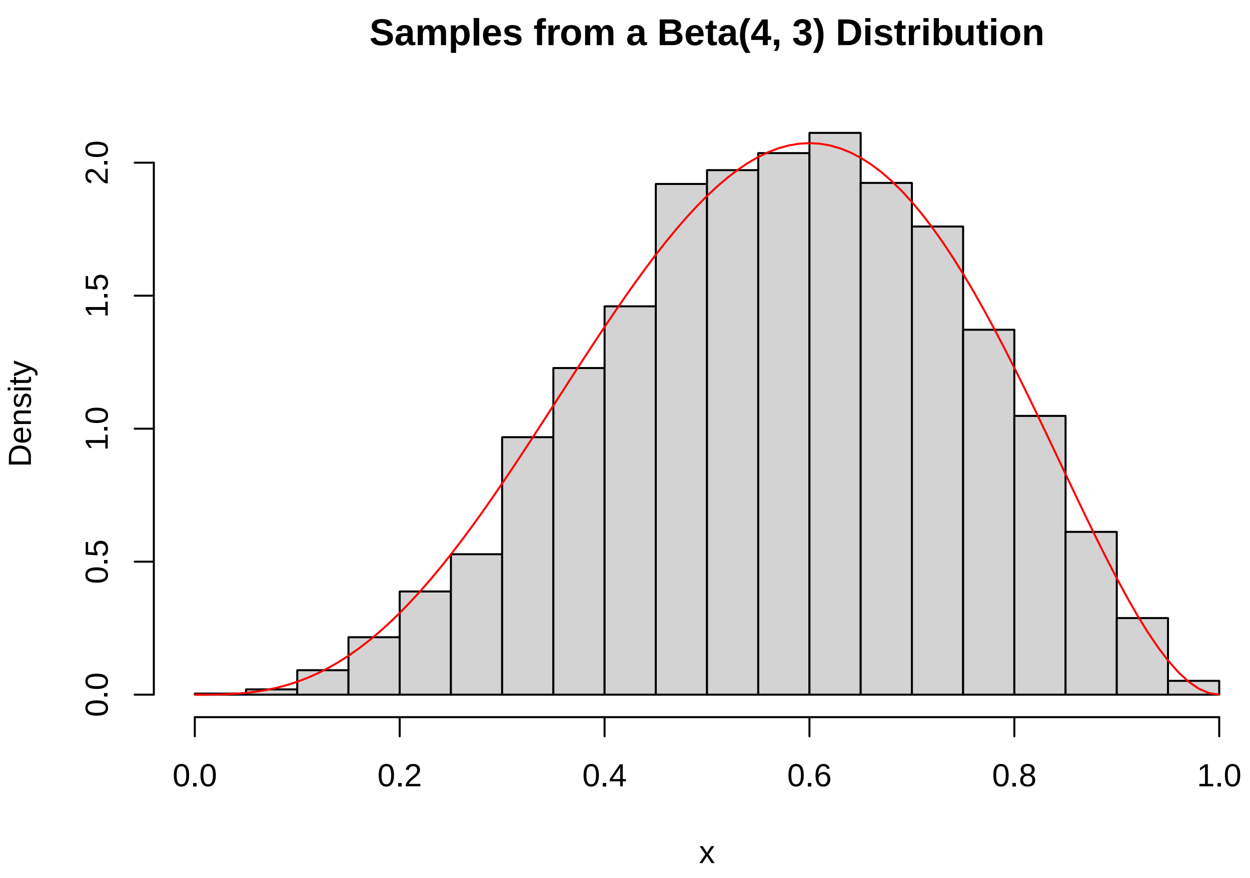
Example 3: Uniform

```
samps <- ars(function(x){dunif(x, 10, 15)}, n = 5000, bounds = c(10, 15), x_init = 11)
hist(samps, xlab = "x", main = "Samples from a Uniform(0, 1) Distribution",
     freq = F)
curve(dunif(x, 10, 15), col = "red", add = T)
```



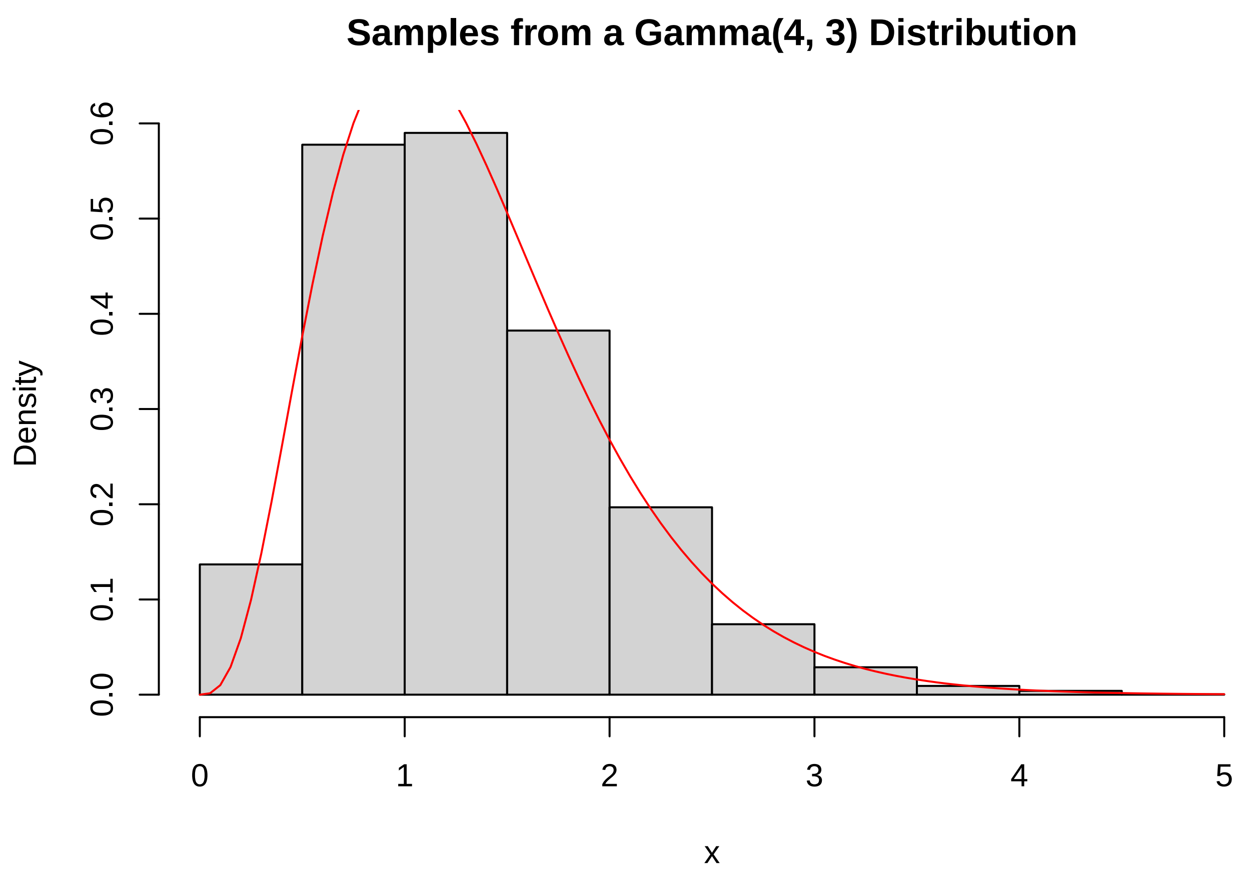
Example 4: Beta

```
samps <- ars(function(x){dbeta(x, 4, 3)}, n = 5000, bounds = c(-2, 2))
hist(samps, xlab = "x", main = "Samples from a Beta(4, 3) Distribution",
     freq = F)
curve(dbeta(x, 4, 3), col = "red", add = T)
```



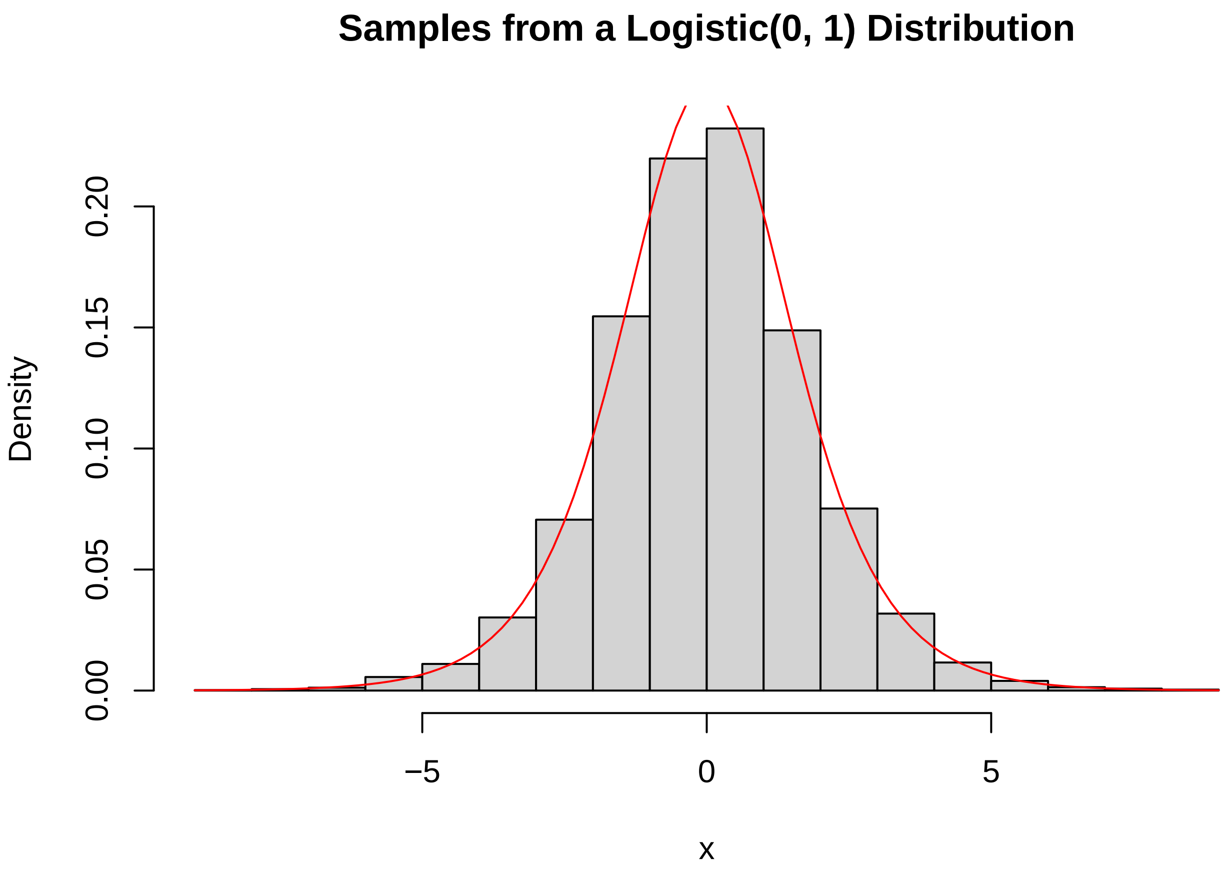
Example 5: Gamma

```
samps <- ars(function(x){dgamma(x, 4, 3)}, n = 5000, bounds = c(0, Inf),
             x_init = 1)
hist(samps, xlab = "x", main = "Samples from a Gamma(4, 3) Distribution",
     freq = F)
curve(dgamma(x, 4, 3), col = "red", add = T)
```



Example 6: Logistic

```
samps <- ars(dlogis, n = 5000)
hist(samps, xlab = "x", main = "Samples from a Logistic(0, 1) Distribution",
     freq = F)
curve(dlogis(x), col = "red", add = T)
```



Contributions

We found a way to evenly distribute the tasks among all of the group members. While below describes a general breakdown of the tasks, every group member helped in some way for every task, whether it be debugging functions or helping to provide ideas when others were stuck. The breakdown is as follows:

- Jeffrey Kuo worked with Hsiang-Chuan Sha in coding the `sample_sk()` and `calc_prob()` functions, thus also completing the sampling step. He also coded the `calc_z()` function and created the R package, as well as assisted in completing the methodology for the sampling step in the write-up.
- Hsiang-Chuan Sha worked with Jeffrey Kuo in coding the `sample_sk()` and `calc_prob()` functions, thus also completing the sampling step. He also wrote the test cases for all of the helper functions and coded the `u()` and `l()` functions.
- Tessa Weiss coded the `check_log_concave()` and `initialize_abscissae()` functions and completed the initialization and updating steps, as well as wrote the test cases for the main `ars()` function. She was also responsible for documentation (comments, formatting, etc.) and writing the majority of the write-up.

References

[1] Gilks, W. R., & Wild, P. (1992). Adaptive rejection sampling for Gibbs sampling. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 41(2), 337-348.

[2] Glen, S. (2022, May 16). Kolmogorov-Smirnov Goodness of Fit Test. Statistics How To. Retrieved December 15, 2022, from https://www.statisticshowto.com/kolmogorov-smirnov-test/

[3] Kolmogorov-Smirnov Goodness-of-Fit Test. 1.3.5.16. Kolmogorov-Smirnov goodness-of-fit test. (n.d.). Retrieved December 15, 2022, from https://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm