STAT 243 - Final Project
Adaptive Rejection Sampler

Keqin Cao (Angela), Mark Campmier, Colleen Sun

December 18, 2020

1 Group GitHub Repository

`https://github.com/sunsgneckq/243_final_project`

2 Introduction

Simulation studies are key to understanding the underlying properties of a statistical model or method. Typically simulation studies use synthetic data generated from a set of random numbers using an existing understanding of the underlying model of the distribution and iterate over the model to approximate an expected value. In the most simple case, these simulation data sets can be directly compiled from a known underlying cumulative density function (CDF) of the sample data. However, often the CDF may not be directly invertible or may poorly characterize the sample data, especially near the tails. To address this shortcoming of the Inverse CDF method, the Rejection Sampling method can be applied (RS). RS uses a function of known characteristics (g(x)) to constrain the CDF to a constant of integration. Then, an envelope function defined as greater than g(x) for all x, can be used to draw random numbers, and the chosen values are accepted or rejected by evaluating with a random number selected from the uniform(0, 1) distribution. Unfortunately, RS can be computationally expensive since it requires optimization to locate the supremum of g(x), and may lead to many rejected values. To compensate for these drawbacks, Adaptive Random Sampling (ARS) was developed.

In this study, the ARS algorithm as described in Gilks and Wild 1992 was developed in the R programming language. Throughout development, the authors observed scientific programming concepts including type assertions, concise documentation, vectorized operations, and unit testing to maintain reproducibility and interpretability of the software. This paper will briefly summarize the relevant aspects of ARS, describe in detail the primary function used to deploy the algorithm as well as its dependencies, and finally explore tests of the statistical properties of the R ARS deployment.

3 Methodology

The main R function to implement ARS, is called ars. It accepts four named inputs: f, N, x0, and bounds, and returns the sample results from the ARS algorithm. The first parameter f must be a vectorized function which returns the PDF of the input values. The most simple input case would be the base R function dnorm, to find the normal distribution density of the input. However, it can be any distribution function including a custom function provided it is vectorized. The second input N is the number of valid samples to generate with ARS. It must be an integer value greater than 0. The third input x0 are the starting values in the sampling domain. It should be a numeric vector of length greater than 0, and is set to the default value of c(-1.0, 1.0). The final named input bounds is the boundaries of the underlying sampling distribution. It should also be a numeric vector of length 2 and is set to the default value of c(-Inf, Inf). Following parameter type assertions, the ars function is be split into three parts: preparation, initialization and sampling. Unless otherwise specified all helper functions are defined in the file helper.R.

3.1 Preparation

The preparation phase begins with type assertions, and basic sanity checks on the input values using the assert_that function from the R package assertthat. The assertions check to ensure f and N are not missing and that they are a function and numeric input of length 2 respectively. Additionally, the assertions check bounds to ensure it is a numeric vector of length 2, and the lower bound is less than the upper bound. If the bounds are set to the same value, the function does not fail, but rather resets the variable to the default bounds value of c(-Inf, Inf) and issues a simple warning message. Finally, the assertions ensure that x0 must be within the bounds. Most of the functions used in the assertions use the base R "is" family of functions (ie. is.numeric(bounds[1])), or simple logic (ie. bounds[1] < bounds[2]). The helper function bound_length_2 is defined within the ars function, and used for the length of 2 assertions for the x0 and bounds inputs as well as the corresponding on_failure message.

After the inputs pass all the assertion statements, intermediary variables are defined for usage in the primary loop. As described in Gilks and Wild, the initialization step consists of defining the piecewise upper and lower hulls, analogous to the envelope and squeeze functions in conventional RS. These components of the rejection envelope are built by finding the tangent line to the log-transformed PDF at k defined points. The hull functions are updated after each rejected sample to reduce the probability of a future rejected samples. To define the "broken curves" expected from the hulls, information on the derivative of the log transformed function is necessary. To this end, the helper function h is defined as the log of f over the defined input x, and the variable h0 as the output of h(x0). The variable h0 cannot be NaN or infinite, either of which will return an error from an assert statement. The variable dh0 stores the output of the numerical derivative of the function h using the finite difference method following the form:

$$dh0 = \frac{h(x0 + dx) - h0}{dx} \tag{1}$$

Where the variable dx represents the step, defined here as 10^{-8}. Invalid values (-Inf or Inf) from x0, h0, and dh0 are removed using the helper function finite_check. The storage variable x_k, h_k, dh_k, and x are defined as empty vectors to be used in the primary loop. Finally, the max number of iterations to allow in the primary loop is set as 1500 with the variable max_iter.

3.2 Initialization

Gilks and Wild describe the initialization step and update step as bookending the algorithm as the first and final steps respectively, however since a while loop is inherently repetitive, they have been condensed into a single initialization step here. The initialization step starts by open the primary loop of the algorithm, a while loop which runs until the length of storage vector x matches N, the input number of samples. Here x will store the accepted values. First, x_k, h_k, and dh_k are appended with the values of x0, h0, and dh0 and sorted in ascending order. Next, x0, h0, and dh0 are all reset to empty vectors. Generating the hull functions from these values may result in discontinuities if there are sequentially repeated values in dh_k or x_k. To account for repeated values, if the length of dh_k is greater than 1, a nested while loop starts which runs until the following criteria is met for at least one indexed value:

$$i = 1, ..., n-1, j = i+1 \tag{2}$$

$$|dh_k[i] - dh_k[j]| > \epsilon \tag{3}$$

$$x_k[j] - x_k[i] > dx \tag{4}$$

Where ϵ is a variable set to the value 10^{-7}, and the variable dx remains its aforementioned value. In the nested while loop, the helper function duplication_check is used to create an index of singly occurring values and the first occurrence of a sequentially occurring value by comparing the difference of each element and the next element to eps or dx similar to the criteria used in the nested while loop initialization. This index is built using x_k, h_k, and dh_k and applied to uniformly decrease or maintain the size of each vector. The while loop is explicitly broken if the length of the vectors is 1 to avoid an indexing error in the nested while loop statement.

As a final check before generating the hull functions, the helper function log_concavity_check is called on dh_k. It will return an error if any value in the following criteria:

$$i = 1, ..., n-1, j = i+1 \tag{5}$$

$$dh_k[i] - dh_k[j] \geq \epsilon \tag{6}$$

After all tests are passed, and sequentially repeated values removed, the hull functions are generated with the numeric variable vectors z_k, u_k, l_k, and s_k. The variable z_k stores the intersections for every x_k the upper hull equation. The variables u_k and l_k store the slope and ordinate intersection at every x_k as a list with the variables m and b for the slopes and intersections respectively. Finally, the variable s_k stores a list of the normalized beta and weights. These values are crucial since x will be drawn from s_k using the inverse CDF method in the sampling phase. Following equations (1), (2), (3), and (4) from Gilks and Wild, the helper functions intercept_z_j, slope_intercept_z_j, slope_intercept_l_j, and beta_u_x are used to calculate z_k, u_k, l_k, and s_k respectively.

The intercept_z_j function accepts x_k, h_k, dh_k, and bounds as inputs. It returns the vector z_k starting with the first and last element assigned to the first and second element of bounds, and the intervening values filled by the following equation:

$$i = 1, ..., n-1, j = i+1 \tag{7}$$

$$z_k = \frac{h_k[j] - h_k[i] - x_k[j] \times dh_k[j] + x_k[i] \times dh_k[i]}{dh_k[i] - dh_k[j]} \tag{8}$$

The slope_intercept_z_j function accepts x_k, h_k, and dh_k and returns the list of vectors, u_k. The list contains the vector m set to dh, and b which follows the form:

$$b = h_k - x_k \times dh_k \tag{9}$$

The slope_intercept_l_j function accepts x_k and h_k as input, and returns the list of vectors, l_k. The list of vectors is composed of the vectors m and b, following the forms:

$$i = 1, ..., n-1, j = i+1 \tag{10}$$

$$m = \frac{h_k[j] - h_k[i]}{x[j] - x[i]} \tag{11}$$

$$b = \frac{x_k[j] \times h_k[i] - x_k[i] \times h_k[j]}{x_k[j] - x_k[i]} \tag{12}$$

Finally, the beta_u_x function accepts the m vector from u_k, the b vector from u_k, and z_k. It returns s_k as a list of vectors beta, and w. The vector beta is calculated following the form:

$$summation_z = \begin{cases} \exp(b) \times (z_k[j] - z_k[i]), & \text{if m[i] == 0} \\ \frac{\exp(b)}{m} \times (\exp(m \times z_k[j]) - \exp(m \times z_k[i]), & \text{if m[i] != 0} \end{cases} \tag{13}$$

$$\beta = \frac{\exp(b)}{\sum summation_z} \tag{14}$$

The summation in the denominator of above expression is the numerical approximation of the integral from Gilks and Wild equation (3). To avoid an undefined error, separate approaches were used to compute the value value as described in the summation_z variable above, composed of two variables in the R code as non_zero_boolean, and summation_z for non-zero m values and zero m values respectively. If any value of s_k is zero, the function beta_u_x will throw an error, with the message 'Area of s(x)=exp(u(x)) <= 0.' The second vector in the list, w is calculated using similar logic to beta, following the form:

$$weight = \begin{cases} \beta \times (z_k[j] - z_k[i]), & \text{m[i] == 0} \\ \frac{\beta}{m} \times \exp(m \times z_k[j]) - \exp(m \times z_k[i]), & \text{m[i] != 0} \end{cases} \tag{15}$$

$$w = \begin{cases} weight, & \text{if weight} > 0 \text{ and weight is not } -\infty \text{ or } \infty \\ 0, & \text{if weight == 0 or weight is } -\infty \text{ or } \infty \end{cases} \tag{16}$$

3.3 Sampling

The first step of the sampling phase is to sample x using by using the helper function sampling_x. The function sampling_x takes the size of the the chunk, the vector beta from the list s_k, the vector m from the list u_k, the vector w from the list s_k, and the vector z_k and returns the list of vectors also called sampling_x. The sampling_x function uses the base R function sample to generate a random vector of valid indices of u_k$m. The sample function takes the index of u_k$m as the set to draw, the chunk size as the number of samples, and the weights as the probability of drawing each index of u_k$m as inputs. The output of sample is set to the intermediary variable sampling in the helper function, and used to generate the other vector of the sampling_x list, x. The inverse CDF method is applied here to generate x using the following form to account for sampled zeros:

$$i = \text{sample(1:length(w), chunk\_size, replace = True, prob = w)} \tag{17}$$

$$x = \begin{cases} z_k + w \times \frac{\text{runif(chunk\_size)}}{\beta}, & \text{m[i] == 0} \\ \frac{1}{m} \times \log(\frac{m \times w}{\beta} \times \text{runif(chunk\_size)} + \exp(m) \times z_k), & \text{m[i] != 0} \end{cases} \tag{18}$$

The final output of the function sampling_x is the list sampling_x consisting of the vector of inverse CDF sampled x values, x_s, and the vector referenced to as i in equation (17), and stored as J.

As in the conventional RS algorithm, a vector of randomly generated numbers from a uniform distribution bound [0, 1] is required to evaluate the rejection criteria. The values are generated in the primary loop using the base R runif function with the output vector size set to the current chunk size (or number of samples left) and stored in the variable w. Next, invalid values are removed before evaluating the rejection criteria. All values outside of bounds are removed from x_s, w, and sampling_x$J using the helper function boundary check to index the points outside of the boundaries (from the variable boundary), and issues a warning using the helper function boundary_warning with the message 'Check bounds! The sampled x is outside bounds.' Now that the sampled values are properly formatted, the first rejection test can be performed using the following form:

$$boundary = \begin{cases} 0, & \text{if l\_k(x) == } -\infty \\ 1, & \text{if l\_k(x) } > -\infty \end{cases} \tag{19}$$

$$i = \begin{cases} J-1, & \text{if x\_s < x\_k[J]} \\ J, & \text{if x\_s} \geq \text{x\_k[J]} \end{cases} \tag{20}$$

$$y = \exp(x_s[boundary] \times (l_k\$m[i[boundary]] - u_k\$m[J[boundary]]) + l_k\$b[i[boundary]] - u_k\$b[J[boundary]]) \tag{21}$$

$$\begin{cases} \text{Accept}, & w[boundary] \leq y \\ \text{Reject}, & w[boundary] > y \end{cases} \tag{22}$$

Here the index boundary, stored in the ars function as boundary, is a modified version of J to account for l_k values of $-\infty$ and filter them out before testing. Next, the index J is reset based on the newly sampled values, x_s, and the previously sampled values, x_k, according to the indexing shown in equation (20). Then, as outlined in equations (21) and (22), following the description presented by Gilks and Wild's first rejection test. The accepted values are appended to the variable x, and the rejected variables are appended to x0. The final step of the primary while loop used in this implementation of the ARS algorithm is the second rejection test. The second rejection test is only implemented if the length of x0 is larger than 0, that is there are rejected values in a given iteration since x0 is reset each iteration of the loop in the preparation phase. First the log of f evaluated at x0 is evaluated and stored as h0. Next the numerical derivative of h0 is calculated, as demonstrated in equation (1), and set to the variable dh0. All NaN and infinite values are removed from x0, h0, and dh0 uniformly similarly to the previously described methodology in the preparation phase using the helper function finite_check. Finally, x is appended with the values of x0 which pass the second rejection test following the criteria here:

$$\begin{cases} \text{Accept}, & w \leq \exp(h0 - x0 \times u_k\$m[J] - u_k\$b[J]) \\ \text{Reject}, & w > \exp(h0 - x0 \times u_k\$m[J] - u_k\$b[J]) \end{cases} \tag{23}$$

Once the primary while loop terminates upon successfully filling the x vector with enough samples as required by the input parameter N, all valid x values are returned by the variable res, a vector of length N.

4 Results

4.1 Testing

All unit testing followed the usage of the R package testthat, such that the testthat function test_package('ars') could be used to streamline testing. The following subsections address the major topics utilized to test the ars function in the args.R file. Additional testing for the helper functions were also produced, but primarily address input sanity checks, so this section will only focus on the main function since the helper functions must work to pass these tests. The tests can be found in the file test-ars.R in the tests directory.There are a total of 47 tests.

4.1.1 Input Checks

This series of tests ensures that the ars function returns an error when the two mandatory names arguments f, and N are missing. It also tests the input types and options. Tests include ensuring f is a function, N is numeric, N is greater than 0, x0 is numeric, bounds is a numeric vector of length 2, bounds are sequentially ordered, and that x0 is contained within the bounds.

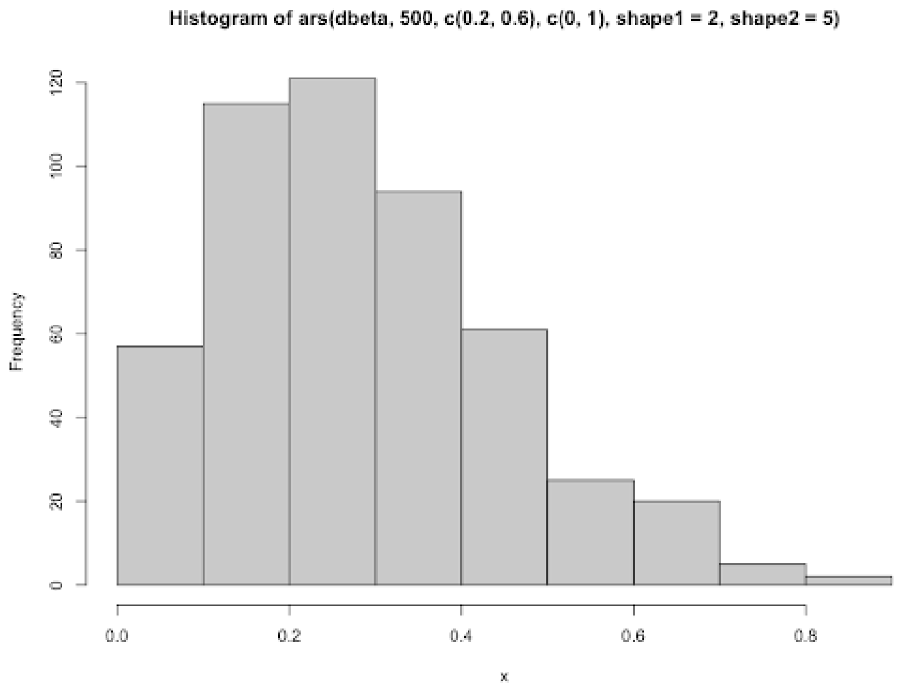
4.1.2 Log-Concavity and Poorly Defined Functions

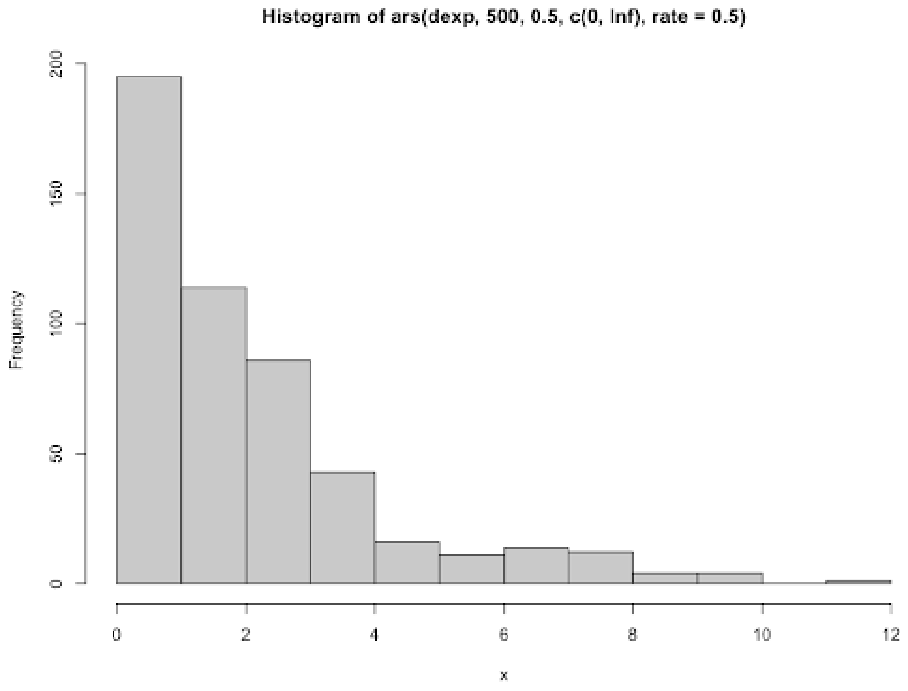
The log-concavity checks are vital to the ARS algorithm. Here, known non-log concave density functions are input a tested to ensure ars throws an error. Tested functions include dcauchy, dlnorm, and dpareto. Additionally, functions which are log concave but have incorrectly formatted inputs resulting in non-log concavity or undefined values were also tested. This includes an exponential distribution with negative bounds values, and a normal distribution with an incorrectly formatted x0.

4.1.3 Distribution Tests

The following distributions were tested to ensure the ars function returned mean values within the 95% confidence interval when tested with a set random seed: normal, exponential, uniform, gamma, beta, pareto, and laplace. Note that the pareto and laplace distributions are not in base R, but are from the EnvStats and extraDistr packages respectively. Without all the necessary packages installed, the pareto and laplace tests will return errors.

4.2 Distribution Examples Gallery





5 Summary

Adaptive Random Sampling is a computationally efficient technique relative to conventional RS for log-concave functions. This study used the R programming language to develop functional software with vectorized operations, type assertions, and thorough unit testing. The primary function, ars, successfully sampled a variety of functions and threw informative exceptions and warnings for improperly or poorly posed usage. Overall, this study demonstrated how best practices in statistical computing can be applied to build software relevant to common problems found in simulation studies.

6 Author Contributions

All group members contributed to the ars function design, helper functions, debugging, testing, documentation, and write-up. One group member was assigned the role of uploading certain tasks to the GitHub repo to avoid accidental duplication or mismanagement of branches. KC was put in charge of the skeleton of the package, and lead the design of the ARS algorithm implementation. CS was responsible for the testing files, debugging, and ensuring uniformity of documentation. MC was responsible for editing, and uploading the write-up files. Collaboration took place over joint coding/writing sessions on Zoom, and Google Hangouts. Manuscript assembly was jointly performed using Overleaf.

7 References

Gilks and Wild. *Adaptive Rejection Sampling for Gibbs Sampling* (1992)
Paciorek. *STAT243 Unit 9 - Simulation* (2020)