STAT 243 Final Project: Adaptive Rejection Sampling

Kristoffer Hernandez, Leon Weingartner, Nikita Mehandru

2022-12-14

```
library(ars)
library(testthat)
```

Introduction

Adaptive rejection sampling (ARS) can be useful when evaluation of the probability density function is computationally expensive. ARS evaluates a small number of samples from h(x), which is the log of the probability density function of interest, f(x). ARS rests on two assumptions, that the distribution be: 1) log-concave, and 2) univariate. As sampling proceeds, the rejection envelope and the squeezing function, both piecewise exponential functions, converge to the density function.

The R package, ars, containing our solution can be found using the following link: https://github.berkeley.edu/khern045/ars

Methodology

Following the logic from Gilks et al. (1992), we begin by initializing the abscissa within some set of bounds. Before doing so, we implement two initial checks to ensure that parameter f is a valid function and that it is numeric. We randomly sample to obtain our first point. To calculate our second and third points, we search to the left of the first point where the derivative is greater than zero and to the the right of the first point where the derivative is negative, respectively.

To calculate the envelope function, we determine the upper and lower bounds for $h(x)$ from a distribution from 0 to infinity. The upper bound is based on the tangent line between the intersections of the x-values and $h(x)$ while the lower bound is the line between the x-points.

We calculate the Zs following the formula in the paper. To sample from the exponential upper hull, for every Z, we calculate the areas between zi and $zi + 1$. We then choose an interval index with probabilities proportional to their areas and implement a function to calculate the inverse cdf of a randomly sampled uniform probability, with the areas of the intervals before that specific interval as normalizing constants.

For each point (x), we use the squeezing and rejection test to determine whether the points is accepted of rejected. If x passes the squeezing test formed by the upper and lower hull functions, then it is accepted at the squeezing step and will not proceed further. If x fails to pass the squeezing test, it will proceed to the rejection test formed by the h density function and the upper hull function. If x passes the rejection test, it will be accepted. Otherwise, x will be rejected. The while loop ("while(length(samples) < N)") allows us to repeatedly sample x until we hit the sample size we want with a set of accepted x points.

ars() Code Overview

We first ensure that the given parameters are valid and throw and error otherwise. This enhances the usability of the the library package so we don't get any unexpected errors when computing the ars.

```
##########PARAMETER CHECKS############
 assertthat::assert_that(is.numeric(bounds), is.vector(bounds))

 assertthat::assert_that(is.numeric(N))

 assertthat::assert_that(is.function(f),
             msg = paste("Error: parameter f is not a valid function!"))

 if (bounds[1] == bounds[2] || bounds[1] > bounds[2] || length(bounds) != 2) {
   stop('ERROR: invalid bounds')
 }
 ######################################
```

We use sub functions to calculate the log density and derivative of the log density at x.

```
 h <- function(x) {
   #return (log(f(x, ...)))
   return(log(f(x, ...)))
 }

 h_prime <- function(x) {
   return (numDeriv::grad(h, x))
 }
```

Before we begin sampling, we initialize our samples vector and set an initial abcissa along with the corresponding h(x) values and derivatives. Our initial abcissa includes values of positive and negative slopes where we will calculate the intersection within our algorithm as described above in the methodology section.

```
samples = c()

 #write function to chose abscissae within bounds
 vars <- set_abcissa(f, h, h_prime, bounds, ...)

 xs <- vars$xs
 hs <- vars$hs
 h_primes <- vars$h_primes
```

The algorithm when excluding details for each helper function is fairly simple where it runs on a while loop until a specified N samples have been reached. Our sample step function samples an x relative to the density of the envelope. We choose to accept or reject the sample and update the appropriate vectors so that the next iteration can update the envelope density.

```
while(length(samples) < N){

   #SAMPLE STEP

   x = sample_step(xs, hs, h_primes, bounds)
```

```
    ux = u_k(x, xs, hs, h_primes, bounds)
    lx = l_k(x, xs, hs)

    u = runif(1)

    #1st rejection step (squeeze test)
    if(u <= exp(lx)/exp(ux)){
      #accept sample
      samples <- append(samples, x)
    }
    #2st rejection step
    else if(u <= exp(h(x))/exp(ux)){
      #accept sample
      samples <- append(samples, x)

    }
    else{
      #reject sample

    }

    #update xs, hs and h_primes
    xs <- append(xs, x)
    xs <- sort(xs)
    i = which(xs == x)
    hs <- append(hs, h(x), after = i-1)
    h_primes <- append(h_primes, h_prime(x), after = i-1)
  }


  return(samples)
```

a) Modularity

Drawing from best practices in Unit 4 lecture notes, our group tried practicing defensive programming. In addition to implementing checks that would yield errors to the user, we wrote modular code for core functions, including the initialization of points, calculation of the lower and upper hulls, and the sample step.

b) Testing & Examples

In addition to the input function tests checking the parameter f function, we impose checks when initializing the abscissa. Specifically, we test that the bounds are not equal to one another, that the lower bound is not greater to the upper bound, and that only two bounds are inputted. To optimize the function, we check that it does not take longer than 10 seconds to identify the points.

In our formal testthat file, we conduct a number of formal checks. First, we check that our ars outputs are producing the correct amount of samples. Second, we check for cases when the input is not a function, which yields an error to the user. Lastly, we calculate the density plot for a variety of log-concave distributions to ensure that the the probability density sums to 1.

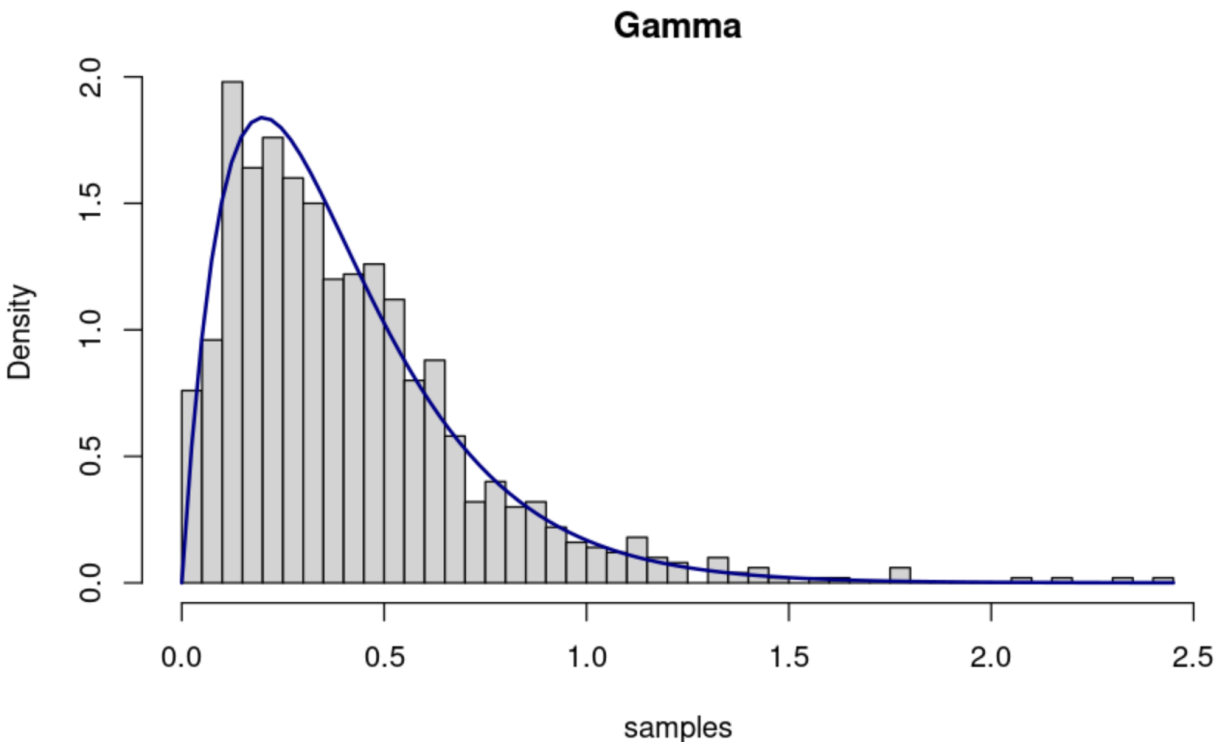
Running From a Fresh Install

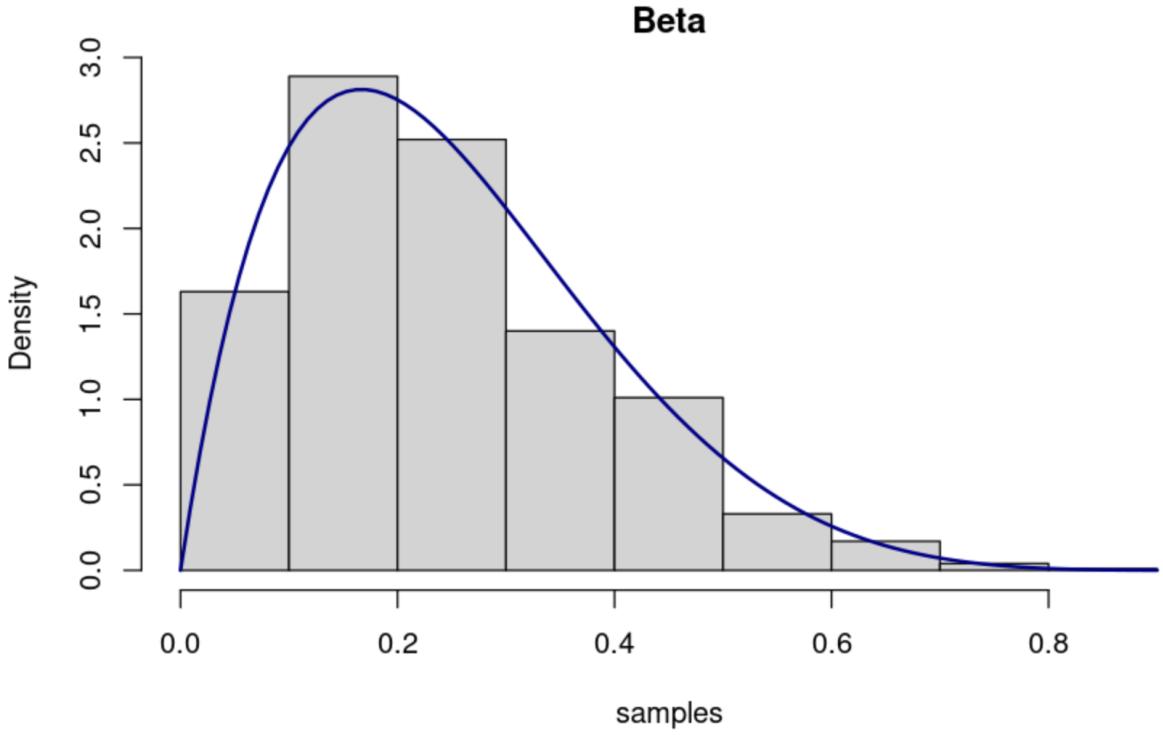
In order to run our unit tests from a fresh install of the ars package, follow the below steps in your **terminal**. We ran 12 tests to verify that errors are outputted if incorrect parameters are inputted, the length of the samples are consistent, and that our sample distribution areas integrate to 1.

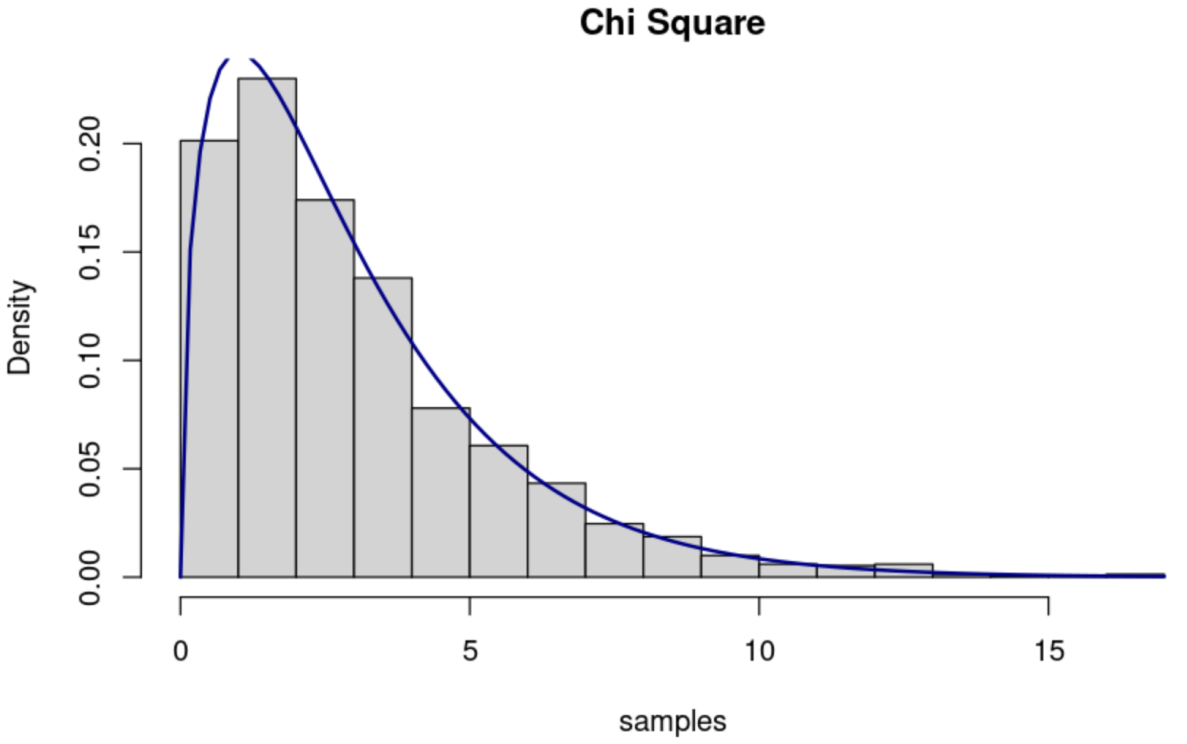
```
git clone https://github.berkeley.edu/khern045/ars.git
cd ars
R
devtools::load_all()
testthat::test_file('./tests/testthat/test-ars.R')
# testthat::test_package("ars")
# the above comment is an alternative to testthat::test_file('./tests/testthat/test-ars.R').
#Both functions will work.
```

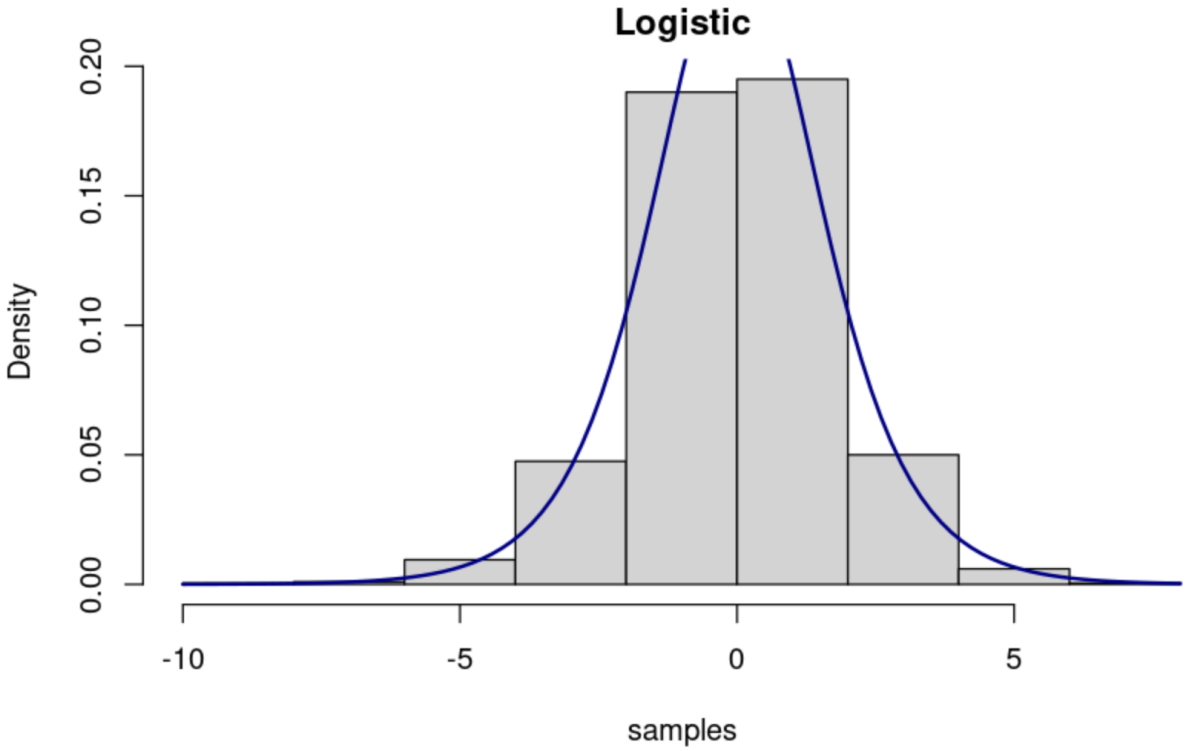
Adaptive Rejection Samples Examples and Associated Distribution Curves

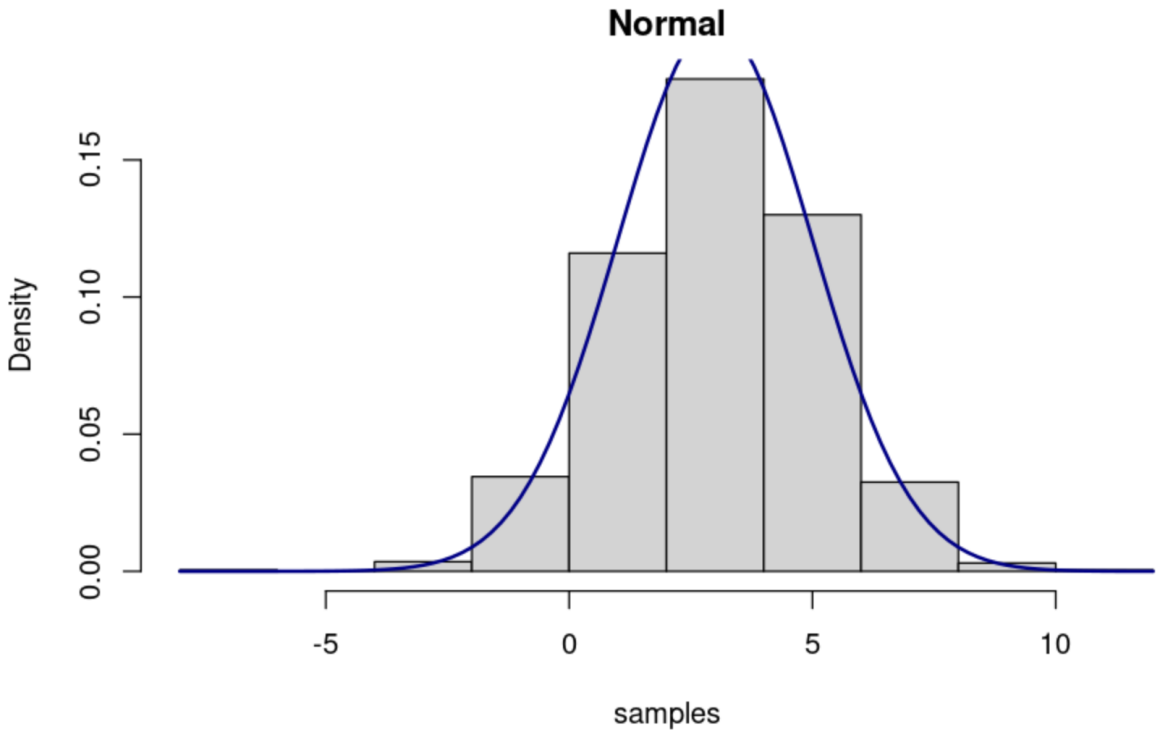
```
library(ars)
N <- 1000
samples <- ars(dgamma, N, bounds=c(0,Inf), shape = 2, rate = 5)
hist(samples, prob= TRUE, main= "Gamma", breaks = 50)
curve(dgamma(x, shape = 2, rate = 5),
     col="darkblue", lwd=2, add=TRUE, yaxt="n")



samples <- ars(dbeta, N, bounds=c(0,1), shape1 = 2, shape2 = 6)
hist(samples, prob= TRUE, "Beta")
curve(dbeta(x, shape1 = 2, shape2 = 6),
     col="darkblue", lwd=2, add=TRUE, yaxt="n")



samples <- ars(dchisq, 1500, bounds=c(0,Inf), df = 3)
hist(samples, prob= TRUE, main = "Chi Square")
curve(dchisq(x, df = 3),
      col="darkblue", lwd=2, add=TRUE, yaxt="n")


samples <- ars(dlogis, N, bounds = c(-10, 10))
hist(samples, prob= TRUE, main = "Logistic")
curve(dlogis(x),
      col="darkblue", lwd=2, add=TRUE, yaxt="n")


samples <- ars(dnorm, N, bounds = c(-20, 20), mean = 3, sd = 2)
hist(samples, prob= TRUE, main = "Normal")
curve(dnorm(x, mean = 3, sd = 2),
      col="darkblue", lwd=2, add=TRUE, yaxt="n")
```


Gamma
Density
0.0
0.5
1.0
1.5
2.0
0.0
0.5
1.0
1.5
2.0
2.5
samples


Beta
Density
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0.0
0.2
0.4
0.6
0.8
samples


Chi Square
Density
0.00
0.05
0.10
0.15
0.20
0
5
10
15
samples


Logistic
Density
0.00
0.05
0.10
0.15
0.20
-10
-5
0
5
samples



Contributions

Our team worked together to split the tasks in an efficient manner. We often implemented proper coding practices that utilize a driver (person at the keyboard) and 2 navigators to plan and steer the project in the right direction. The following is a breakdown of how the tasks were completed by each team member:

Kristoffer Hernandez ~ Responsible for debugging ars function, vectorizing code, assembling R package, testing file

Nikita Mehandru ~ Responsible for writeup, debugging helper functions. Contributed test cases and wrote examples

Leon Weingartner ~ Responsible for coding ars and helper functions, Contributed to test cases and debugging

References

#' 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. #' #' Markou, S. (Dec 2022). *Adaptive rejection sampling.* Random walks. #' https://random-walks.org/content/misc/ars/ars.html #' #' Wickham, H. (2015). *R packages.* O'Reilly Media. #' https://r-pkgs.org/testing-basics.html #'