STAT 243 Final Project
Implementation of an Adaptive-Rejection Sampling Method

Ye Zhi, Boying Gong, Max Gardner, Alexander Brandt

December 17, 2015

1 Project URL

`https://github.com/boyinggong/243FinalProject`

2 Theory

Our final project was to implement an adaptive-rejection sampling method in R. Our method largely implements the methods described by Gilks and Wild. ARS attempts to efficiently generate random values according to a user-defined distribution (and a user-defined domain). The method follows a few general steps, exploiting the tendency of many important and useful distributions to be log-concave (even if the distributions themselves are not concave). For some distribution $g(x)$, let us define $h(x) = \log(g(x))$. Note, this point, if $h(x)$ is found to be not log-concave, then our program will reject the values.

Now, in order to efficiently sample from values (and not waste time generating values that will ultimately be rejected), we construct an upper envelope function that bounds $h(x)$. This is performed by taking two or more starting points from within the domain such that they lie within our users defined bounds. At points we will calculate the tangent's slope at x_i. We then calculate the x-intercept where these slopes meet with:

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

Where $j \in [1, k-1]$. Note that z_0 and z_k are the upper and lower bounds of the function, respectively, even if those bounds are $-\infty$ or ∞.

Now that we can define the envelope for $x \in [z_{j-1}, z_j]$:

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

By exponentiating and normalizing this $u_k(x)$, we now have a density from which to sample our random values:

$$s_k(x) = \frac{\exp(u_k(x))}{N}$$

Where N is simply the all-space integral value of $s_k(x)$ (i.e. $N = \int_D s_k(x)dx$).

Furthermore, we exploit a computationally inexpensive procedure to compute a lower hull, which is defined as:

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

$l_k(x)$ can allow us to perform a squeezing test to quickly accept or reject proposed sampling values. This is performed by selecting some value x^t from $s_k(x)$, and drawing a value w from the uniform distribution defined from [0,1]. We can accept x^t if:

$$w \leq \exp(l_k(x^t) - u_k(x^t))$$

Or if:

$$w \leq \exp(h(x^t) - u_k(x^t))$$

These are the squeezing test and rejection test, respectively. As we continue to compute $h(x^t$ and $h'(x^t)$, we can update the points that construct our upper and lower hulls, thereby increasing the chance of new values drawn from $s_k(x)$ being accepted. This is the mathematical and statistical intuition behind the implementation discussed in the next section.

3 Implementation

In order to create a modular solution, we have created several helper functions in various files which are called by ars in ars.R in an order described above. Our code was made modular, portable, and testable by use of functions that define each subroutine in the ARS calculation.

3.1 Main function

```
ars(fun, my_total_range, n, MAX_HOLDING = 1000, nGroup = 1, ...)
```

Besides the basic dendity function *fun*, support *my total range*, sample size *n*, our main sampling function allow the following argument:

- MAX_HOLDING maximum number of points we can include in the hull. When number of points getting larger, both upper and lower hull gets close to the density function, we could stop updating the hull to improve efficiency.
- nGroup number of samples generate at once from the upper hull. Note that here we allow users vectorize the sampling process. The default value is 1, which means to sample sequentially.
- ... allow user to pass arguments of *fun* to the sampling function.

3.2 Auxiliary functions

1. dh.R
   Arguments: an x value to evaluate, a function g
   Returns: $g'(x)$
   The derivative of h is of critical importance throughout this procedure. To implement, we use a dh method that takes advantage of numerical derivation (via numericDeriv in the stats package) to find our tangents.

2. find_init.R
   Arguments: a function g, and a list specifying the range it is defined on
   Returns: a list of two entries corresponding to the left and right starting hull x_i's.
   Though it would always be best for the user, who has a better understanding of the distribution, to supply their initial points, it was our intention that the program perform well even in cases when the user did not. To this end, ars.R begins by calling find_init.R

3. get_zj.R
   Arguments: x values of hull points, y values (h(x)'s) of hull points, slope of the hull points
   Returns: a list of all intersection points (z_j's)
   To compute the intersection between our tangents, we use get_zj.R. It computes all the intersections given various x_i's from the formula given above.

4. upper_piecewise.R
   Arguments: an x to evaluate, the x values of hull points, y values (h(x)'s) of hull points, slope of the hull points, and the total domain of the probability distribution function
   Returns: u(x)
   This uses a vectorized formulation to compute $u(x)$ by using a boolean operation based on x's range, the range of z_i, to compute the correct piece (i.e. only the relevant line segment for a given range will be used to compete $u(x)$.

5. lower_piecewise.R
   Arguments: an x to evaluate, the x values of hull points, y values (h(x)'s) of hull points, and the total domain of the probability distribution function

   Returns: l(x)
   This uses a vectorized formulation to compute $l(x)$ by using a boolean operation based on x's range, the range of z_i, to compute the correct piece (i.e. only the relevant line segment for a given range will be used to compete $l(x)$. (The only difference is that this is the lower chord hull, and that for points between the domains an the first and last x_i, a large negative number returned as the constant value.

6. sk.R
   Arguments: the x values of hull points, y values (h(x)'s) of hull points, and the regions defined by $z_{j=1,\ldots,N-1}$ as well as the bounds of the probability distribution function

   Returns: a list of the non-normalized integral values of each piecewise exponential
   Takes advantage of an explicit form of the inverse cumulative distribution function for an exponential (given that all $u(x)$ are linear, and thus the integral of $\exp(u(x))$ can be computed explicitly (i.e. without use of any integration packages).

From here, we sample using $s_k(x)$. The procedure, as implemented, is fairly straightforward. To begin, we select one of the pieces of the piecewise exponential probability distribution function by normalizing the curve, drawing a random value from the uniform distribution on [0, 1], and choosing the largest segment index such that the total integral value from the lowest domain to that region's upper bound is greater than the randomly drawn value. Then, based on the normalized sub-region we sample a value x^t using the inverse CDF for the exponential (again, this can be explicitly solved), using another random variable u drawn from the uniform distribution on [0, 1]. More explicitly:

$$u = \int_{z_i}^{x^t} exp(k_i(x - x_i) + y_i)dx = \left.\frac{exp(k(x - x_i) + y_i)}{k}\right|_{z_i}^{x^t}$$

Where u is our randomly generated number, x_i, y_i, and k_i are the x value, y value, and slope of the hull point, and z_i is the lower bound of the region of the piecewise exponential we are drawing from. We can now easily solve to find our sampled value of x^t. This value can then be accepted or rejected using the squeezing or rejection tests which is the final step implemented by ars.R. A limit is also placed on how many hull points can exist for a given simulation. Once we have selected enough values (as defined by the user), we return the array of numbers.

4 Testing

Tests were run by using devtools::test() as well as testthat's test_package('ars', 'main').

1. test-dh.R
   Checks that deriatives match their expected output for a few specific cases

2. test-find_init.R
   Checks the multiple scenearios of bounded/unbounded left and right bounds of probability distribution functions, and ensures their derivatives give the correct slopes.

3. test-get_zj.R
   Checks that a test function produces the expected z_j.

4. test-upper_piecewise.R
   Checks for inconsitencies with inputs (i.e. x is not within the domain of the function)
   Checks for length inconsistency between argument lists

5. test-lower_piecewise.R
   Checks for inconsitencies with inputs (i.e. x is not within the domain of the function)
   Checks for length inconsistency between argument lists

6. test-sk.R
   Checks for inconsitency between argument lengths
   Checks that specific cases integrate to the correct values.

7. test-main.R
   Using Kolmogorov-SmirnovTests tests to compare the empirical CDF of the samples and the the actual CDF. sampling from different distributions and ensures that they give a reasonable p-value when compared to their generative function. We set our significance level to 0.05. Checks sampling with a defined nGroup variable (i.e. making each draw more efficent)
   Checks for user input errors
   Checks that functions with incorrect concavity throw errors

5 Results/Examples

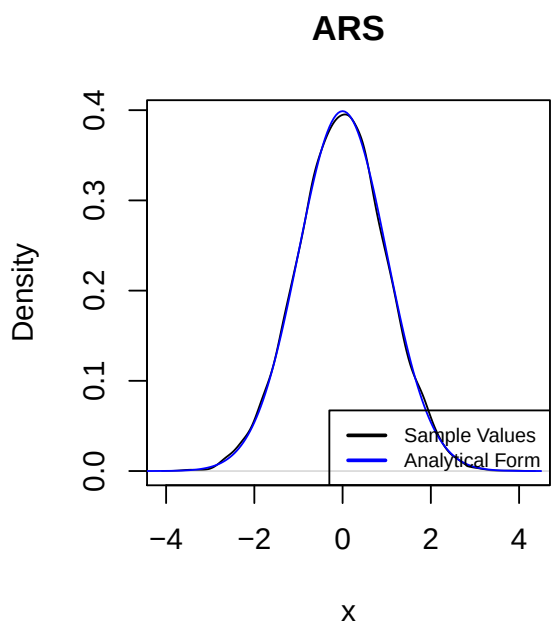
We provide some basic results in addition to the extensive testing of each function individually, which can be seen in more detail in the test suite defined in the package.

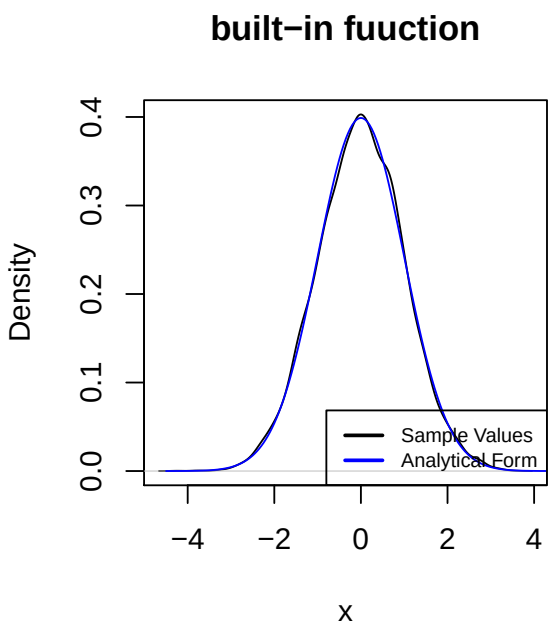
```
install.packages("ars_0.5.tar.gz", repos = NULL, type="source")
library(ars)
```

5.1 Normal Distribution

Standard Normal:

```
par(mfrow = c(1, 2))

sv <- ars(dnorm, c(-Inf,Inf), n = 10000, mean = 0, sd = 1)
plot(density(sv), xlab = "x", main = "ARS")
lines(a <- seq(-4.5, 4.5, by = 0.05), dnorm(a), col='blue')
legend("bottomright", c("Sample Values", "Analytical Form"), col=c("black", "blue"), lwd=2, cex = 0.7)

plot(density(rnorm(10000)), xlab = "x", main = "built-in fuuction")
lines(a <- seq(-4.5, 4.5, by = 0.05), dnorm(a), col='blue')
legend("bottomright", c("Sample Values", "Analytical Form"), col=c("black", "blue"), lwd=2, cex = 0.7)
```

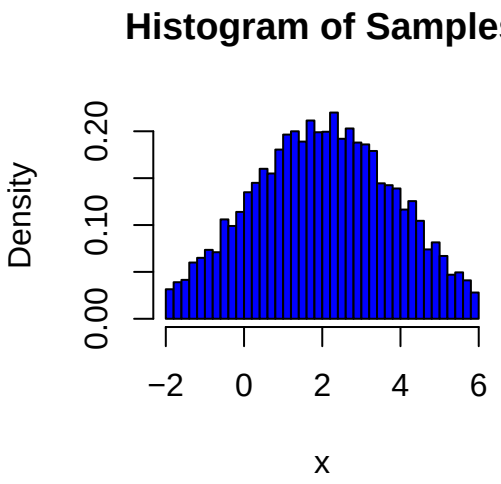





```
## Kolmogorov-Smirnov Tests

ks_res <- ks.test(sv, pnorm)
ks_res_bt <- ks.test(rnorm(10000), pnorm)

## Cramer-Von Mises (CvM) Test

library(goftest)
cvm_res <- cvm.test(sv, "pnorm", mean=0, sd=1)
cvm_res_bt <- cvm.test(rnorm(10000), "pnorm", mean=0, sd=1)

## Anderson-Darling (AD) Test

library(goftest)
ad_res <- ad.test(sv, "pnorm", mean=0, sd=1)
ad_res_bt <- cvm.test(rnorm(10000), "pnorm", mean=0, sd=1)

## summary table

summary <- data.frame(c(ks_res$p.value, cvm_res$p.value, ad_res$p.value),
c(ks_res$p.value > 0.05, cvm_res$p.value > 0.05, ad_res$p.value > 0.05),
c(ks_res_bt$p.value, cvm_res_bt$p.value, ad_res_bt$p.value),
row.names = c("Kolmogorov-Smirnov","Cramer-Von Mises","Anderson-Darling"))
colnames(summary) <- c("ARS p-value","Significant", "Built-in p-value")
print(summary)

##                    ARS p-value Significant Built-in p-value
## Kolmogorov-Smirnov   0.5604256        TRUE        0.5847214
## Cramer-Von Mises    0.4962861        TRUE        0.1204963
## Anderson-Darling     0.5673660        TRUE        0.4923817
```

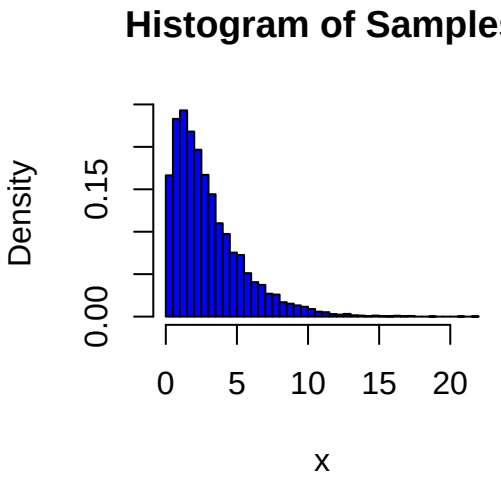
Tail truncated normal:

```
sv <- ars(dnorm, c(-2,6), n = 10000, mean = 2, sd = 2)
hist(sv, col='blue', prob=TRUE, breaks=50, xlab = "x", main="Histogram of Samples")
```



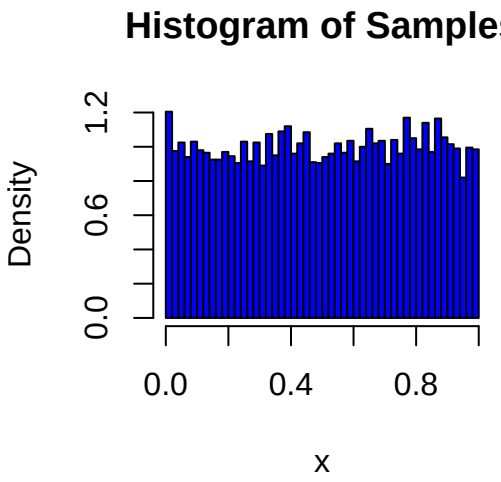
5.2 Chi-square Distribution

```
sv <- ars(dchisq, c(0,Inf), n = 10000, df = 3)
hist(sv, col='blue', prob=TRUE, breaks=50, xlab = "x", main="Histogram of Samples")
```



5.3 Uniform Distribution

```
sv <- ars(dunif, c(0,1), n = 10000, min = 0, max = 1)
hist(sv, col='blue', prob=TRUE, breaks=50, xlab = "x", main="Histogram of Samples")
```



5.4 Throw error when density is not concave

```
ars(dt, c(-Inf,Inf), n = 1000, df = 2)

## Error in ars(dt, c(-Inf, Inf), n = 1000, df = 2):  Please check the log-concavity of probability
density function
```

5.5 Runtime when nGroup varies

We can see from the results that as nGroup gets larger, the runtime drops. This suggests that sampling several points at once do improve the efficiency. However, when nGroup gets too large, the efficiency decreases.

```
system.time(ars(dnorm, c(-Inf,Inf), n = 1000, nGroup = 1))

##    user  system elapsed
##   0.146   0.003   0.148

system.time(ars(dnorm, c(-Inf,Inf), n = 1000, nGroup = 5))

##    user  system elapsed
##   0.108   0.001   0.109

system.time(ars(dnorm, c(-Inf,Inf), n = 1000, nGroup = 20))

##    user  system elapsed
##   0.121   0.002   0.123
```

6 Roll the Credits

Ye Zhi wrote the tests and relative documentations for the unit functions. She is a first-year MA student in Statistics.

Boying Gong wrote the main tests, vectorize and modularize the initial implementation, write the non-log-concave detection and the function to find initial points, expand the function to extreme cases such as uniform, put functions to the package and build the package. She is a first-year MA student in Statistics.

Max Gardner developed auxillary functions for ARS, lead the documentation effort, and rolled everything up into a nice and neat R package. He is a first-year MS/PhD student in Civil Systems.

Alexander Brandt wrote an initial implementation of the ARS and LaTeX'd up the final report. He is a computational biologist interested in worms, humans, and what they have in common (and what they don't!).

7 Citations

1. Gilks, W. R., and P. Wild. "Adaptive Rejection Sampling for Gibbs Sampling". Journal of the Royal Statistical Society. Series C (Applied Statistics) 41.2 (1992): 337–348.

2. Wild, P., and Gilks, W. R. "Algorithm AS 287: Adaptive Rejection Sampling from Log-Concave Density Functions". Journal of the Royal Statistical Society. Series C (Applied Statistics) 42.4 (1993): 701–709.

3. Bagnoli, Mark, and Ted Bergstrom. "Log-concave probability and its applications." Economic theory 26.2 (2005): 445-469.

4. Nadarajah, Saralees, and Samuel Kotz. "Programs in R for computing truncated t distributions." Quality and Reliability Engineering International 23.2 (2007): 273-278.