Stat 243 Final Project

Willy Lai, Steven Chang, Shengying Wang, Jason Poulos

December 12, 2013

Chang, Steven uploaded the code file to bSpace.

In this project, we implement the adaptive rejection sampling algorithm as described in Gilks et al. (1992) in R. Our approach to this problem will follow the method described in section 2.2 of the paper. Our code consists of auxiliary functions that carry out each step of the algorithm, with a main function encompassing all of these functions to carry out the simulation. For example, our auxiliary functions involve finding a set/vector of abscissae, computing the values of z_j, $u_k(x)$, $s_k(x)$, and $l_k(x)$ as described in the paper, sampling from $s_k(x)$, and performing the rejection test. The main function will include all of these functions and a while loop to perform the rejection sampling part of the algorithm. The inputs of the main function are the target density, the Domain on which the density is to be defined on, the number of initial points, and the desired sample size. The output is the vector of our sample as a result of the adaptive rejection sampling algorithm.

How we initialize the abscissae in T_k (pick the values for T_k)

For the initialization of starting points T_k, we start out with two values because this is much easier to choose and is much more manageable in our algorithm. In our function, `startingpoints`, the user has the option to input the upper and/or lower bounds or not. If either the upper bound or lower bound are not given by the user, the upper bound will be ∞ and the lower bound will be $-\infty$. What the starting values are will then depend on whether the domain is bounded or unbounded on either side and if the user has provided any starting values. If the user has provided two starting values, they will be used as our starting points for T_k, regardless of whether the domain is bounded or unbounded on either side. The following discussion assumes the user has not given any starting values.

If D has neither an upper bound nor a lower bound, we pick $a = -4$ and $b = 4$ as the default starting values. Before we assign the starting values as our starting points for T_k, we check if we have an optimal value between these starting values by checking if $h'(a) > 0$ and $h'(b) < 0$. If this is not the case, the function will stop and output an error message, which depends on which inequality was not satisfied, and tell the user the problem and what to do to avoid the error. If a and b stastify $h'(a) > 0$ and $h'(b) < 0$, they will be assigned to T_k as our starting value.

If there is an upper bound but no lower bound, then the upper bound will be one of the starting values, with the other starting value being -4. Before we assign the starting values, we check if the upper bound is less than -4, because this would not make sense for our starting values. If the upper bound is less than -4, then we reassign a as twice the upper bound. If not, then this step is unnecessary. Either way, a and b will be our starting points for T_k.

If there is a lower bound but no upper bound, a similar method is used as for the case where there is an upper bound but no lower bound. The lower bound will be assigned to a, and b will be assigned 4. Next, we check if the lower bound is bigger than 4, and if so we reassign b to be twice of the lower bound. If not, this step is skipped. a and b will then be assigned as our starting points for T_k.

Finally, if D is bounded on both sides, the starting points will be the lower and upper bounds of D.

How we computed the functions $z_j, u_k(x), s_k(x), l_k(x)$

We created the functions $u_k(x)$, `createUpHull`, and $l_k(x)$, `createLowHull`, in R by using the domain D, the set of starting values T_k, and the log of the target density $h(x)$ as inputs. Both functions output a data frame, in which each column represents slopes of the line segments between two abscissae, y intercepts of the line segments, left starting points of each line segment, and right starting points of each line segment. Since $u_k(x)$ and $l_k(x)$ are piecewise linear functions, this method allows us to represent a piecewise linear function in R. The only difference between the outputs of $u_k(x)$ and $l_k(x)$ is that the R code for $u_k(x)$ also outputs the integral of the exponential function raised to the line segment, which is essential for computing $s_k(x)$. The probabilities are then normalized in order to form $s_k(x)$. Also, the code for $u_k(x)$ includes the computation of the intersection points z_j.

How we sampled from $s_k(x)$

We sampled a point from $s_k(x)$ by creating the inverse cdf of $s_k(x)$ and sampling from that. In the function `sampleUp`, we input the data frame from the UpperHull function, which is our $u_k(x)$ function. We then extract out all the probabilities associated with each line segment so that we can employ the Inverse CDF method. By generating a random number w from the Uniform[0,1] distribution and comparing it with the cumulative sum of the probabilities, we pick the line segment corresponding to the first instance w is less than or equal to the cumulative probability. Using the slope and intercept, we can then apply the inverse CDF on w to get a sample candidate point.

How we performed the rejection test:

After sampling a point from $s_k(x)$, we evaluate the upper boundary and lower boundary at this sample point. The function to perform this, `evalSampPt`, takes the data frames for the upper boundary and lower boundary and the sample point as inputs, and returns a vector of $u_k(x)$ and $l_k(x)$ evaluated at the sample point. Afterwards, we take these to points to perform the squeeze test and the rejection test of our sampling test. Our function, `rejectiontest`, will take the values evaluated at `evalSampPt` as well as the sample point and return three logicals, which describe whether we accept or reject the sample point, whether we update our abscissae or not, and whether our target density is log-concave at our sample point. These logicals will decide how we proceed with our algorithm.

How we wrote the whole algorithm into a main function:

After writing all of the above auxiliary functions and steps, we put them together into a main function called `ars`. The inputs for `ars` are the target density g, the domain D with default argument to be a vector of 2 NA's, the two possible starting values a and b, both of which has their default arguments as NA, and the desired sample size n, with 1 as the default argument. The function starts by defining $h(x)$ as the log of the target density and creating an empty numeric vector to be filled with sample points from our algorithm. Next, we build our vector of starting points T_k and data frames representing the functions $u_k(x)$ and $l_k(x)$. The function stops if the starting points T_k are either invalid or outside of the domain.

After creating the starting values and functions, we run a while loop to generate our sample. We created a counter, k, that counts the number of elements in our sample, in which the while loop will run until we have the desired number of elements in our sample. Note that k will only increment by 1 if we accept the sample point. In the while loop, we sample x^* from $s_k(x)$, sample u from a uniform(0,1) distribution, evaluate the sample point x^*, test to see if we should accept or reject the sample point, see if we need to update the starting values T_k, and check if our target density is log-concave. If our target density is not log-concave at the sample point, then the entire function stops and gives an error message telling the user that the density was not log-concave on D. This step ensures that our assumption of log-concavity is satisfied.

If log-concavity is satisfied, we use the logicals for acceptance and update to proceed with our algorithm. We check whether we accept the sample or not before checking if we need to update the vector of initial values, T_k. If acceptance is true, we add our sample, called $x.star$, to the current vector of samples. Then, if update is true, we place the sample, $x.star$, into our vector T_k and sort them in increasing order. If we do not accept the sample, then we update by placing $x.star$ in T_k. We also update the functions $u_k(x)$ and $l_k(x)$ by using the updated T_k. In the process, we check if the updated $u_k(x)$ will give us an error, and if so, the function will stop and print an error message. This step checks if we sampled anything that is too small or too large and if the starting values or domain is valid for our algorithm. Afterwards, the algorithm repeats, with updated values if an update was performed, until we have the desired number of sample points.

The function, `ars`, outputs a vector of n sample points once the sampling algorithm has been completed. This sample should be representative of the target density $g(x)$.

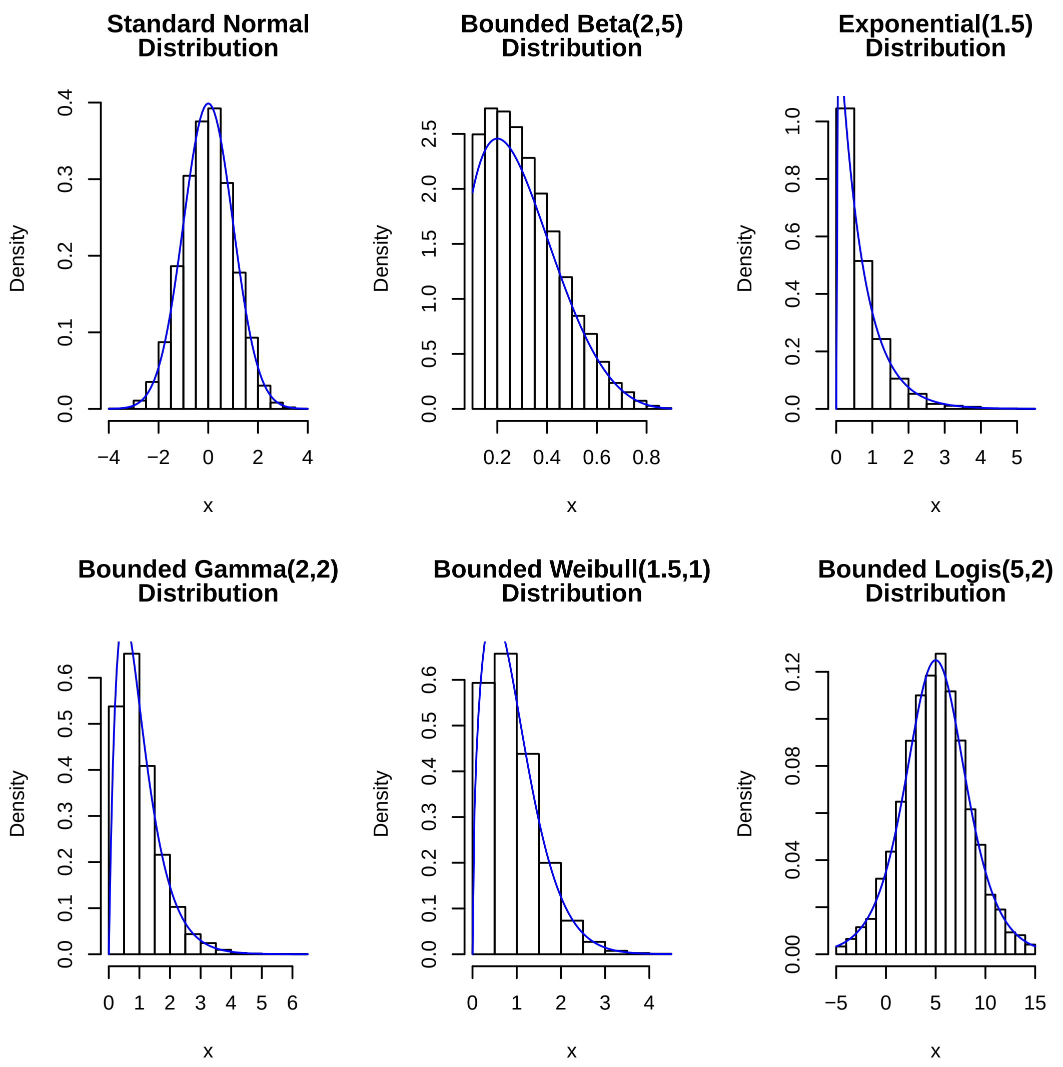
Testing the main function

In our test function, we test whether the `ars` function outputs a sample that is related to the target density the user inputs. For example, if the user inputs the standard normal density as the target density $g(x)$, then `ars` should output a sample that is representative of the standard normal distribution. Thus, we use the Kolmogorov-Smirnoff test to test if the sample generated from the `ars` function matches the distribution described by the target density $g(x)$. If the target density is truncated, then we will compare our `ars` output with a sample from the truncated density (i.e. a two-sample KS test). We will run the test function for six different distributions. For each distribution, the test function will describe what distribution is being used, whether the sample produced by the `ars` function matches the distribution tested as specified by the KS test, and a histogram of the sample with the true density function overlaying the histogram in blue.

```
> source('ars.R')

> test()

[1] "Test a standard normal."
[1] "Pass the Kolmogorov-Smirnov test at level = 0.05."
[1] "Test for Bounded Beta(2,5)"
[1] "Pass the Kolmogorov-Smirnov test at level = 0.05."
[1] "Test an Exponential with Rate 1.5"
[1] "Pass the Kolmogorov-Smirnov test at level = 0.05."
[1] "Test for Bounded Gamma(2,2)"
[1] "Pass the Kolmogorov-Smirnov test at level = 0.05."
[1] "Test for Bounded Weibull(1.5,1)"
[1] "Pass the Kolmogorov-Smirnov test at level = 0.05."
[1] "Test for Bounded Logis(5,2)"
[1] "Pass the Kolmogorov-Smirnov test at level = 0.05."
```



We see that all of our test cases appear to pass the Kolmogorov-Smirnoff test at the 5% level. In addition to that, the histograms appear to match with the true distribution in shape.

Team Roles

Everybody dabbled in every aspect of the project. Steven primarily organized and wrote the main function along with the functions that created the upper hull, lower hull, and sampled from the upper hull. Along with Shengying, he tested the main function along with all the auxiliary functions. Shengying primarily tested all the code that Steven wrote while helping Steven organize the logical flow. He also collaborated with Willy and Jason in writing the function that determines the starting points. Willy wrote the function that performs the rejection and updating tests in addition to collaborating with Shengying and Jason on writing the function that determines the starting values. He was the primary organizer and writer of the written report. Jason collaborated with Willy and Shengying in writing the function that determines the starting points. He was the primary organizer and writer of the help comments.