Stat 243 Final Project

Willy Lai, Steven Chang, Shenying Wang, Jason Poulos

December 10, 2013

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 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 are starting out with two values because this is much easier to choose and is much more manageable in our algorithm. In our function, startingpoints, we initialize the abscissae by taking the Domain D and assigning the first value for T_k with the lower bound of D and the second value for T_k with the upper bound for D. The user has the option to input the upper and/or lower bounds or not at all. 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 $-\infty$. If D has not have a lower bound, an upper bound, or both, we pick -4 and 4 as the default lower and upper bounds, respectively, and assign them as our starting points for T_k. In this case, we have to check if we have an optimal value in this interval, otherwise we would not be able to perform the algorithm, stop the function, and tell the user to pick a better domain. If D does not have a lower bound but has an upper bound, we assign -4 to lower starting point of T_k and check if the lower bound is lower than -4, and if it is, we repick the lower point to be twice of the upper bound. A similar method is performed if D does not have an upper bound but has an upper bound. In this case, we assign 4 for the upper value for T_k. Then we check if the lower bound is larger than 4, and if so, we repick the upper value as twice of the lower bound.

Testing the initialization step:

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

We created the functions $u_k(x)$ and $l_k(x)$ in R by having the user input the domain, the set of abscissae, and the log of the target density. These inputs allow the user to change the these functions whenever any of the inputs change, which is especially useful when updating the abscissae is necessary. 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 $s_k(x)$. Also, the code for $u_k(x)$ includes the computation of the intersection points z_j.

Testing these functions:

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 our code, we input the data frame from the UpperHull function, which is our $u_k(x)$ function.

Testing this sampling

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, will involve taking the data frame for the upper boundary and lower boundary and the sample point as inputs, and returning a vector of the $u_k(x)$ 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. These logicals will decide how we proceed with our algorithm. Most importantly, 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. This step ensures that our assumption of log-concavity is satisfied.

Testing the rejection test

How we updated the values when necessary.
Test the update steps.

How we wrote the whole algorithm into a main function:
Test the Main function performing our rejection sampling algorithm.