————Stat 243 Final Project————
Repository Name: github.com/wilsoncai1992/stat243_2015_project

Weixin Cai (wilsoncai1992), Jianbo Chen (Jianbo1992), HR Huber-Rodriguez (HRHuber), Chenyu Wang (Chenyu-Renee)

December 17, 2015

Outline and Overview

Our final project for Statistics 243 was to create an adaptive rejection sampling (ARS) package in R, a process outlined by W.R. Gilks in his 19992 paper, "Derivative-free Adaptive Rejection Sampling for Gibbs Sampling." The code required to execute this process is inherently modular, and other steps including testing, packaging and plotting helped to distill the project into discrete components that could be worked on simultaneously by various members of the group.

The function itself, titled ars243, takes as mandatory inputs a function and a desired number of sample points, 'n'. It also optionally takes the log of the function (which replaces the function if input), the starting number of Tk points, and the domain of the function. A combination of higher level functions, lower level functions and input checks make up the three components of the code. Checks make sure that the input distribution is log concave, that there are enough initial TK points, and that the domain over which the distribution exists is correct. High level functions check to see if sampled points are squeezed or rejected. They call on lower level functions, which complete tasks such as updating the the TK points, re-evaluating the upper and lower envelopes, and running a binary search to probe which section of these piecewise functions is used to evaluate a particular point.

Individual Functions

The first section of code checks for input validity. It sets the upper and lower bounds as given by the domain (or infinity as default), creates a minimum of 4 initial Tk points, and makes sure the given function is indeed a function, or else it converts it to one, and takes its log, as is required by the procedure. We then initiate our ultimate 'final values' vector and create the grid of abscissa points, equal to the value of the input parameter 'k', given. Points are removed where the derivative is equal to zero, and the initial abscissae is complete.

Next, a check for log concavity is performed on the abscissae grid. If the function doesn't appear to be log concave by checking the derivatives at the left and right end, then the function is ceased and an error message appears. This can also occur if not enough Tk points are given. Now we are into the meat of the code. The 'lupdater' function, a lower level function that draws the required chords of the lower envelope, is called once. It places the slope and the y intercept in a two column matrix, with rows equal to the current number of Tk points. New coefficients are returned every time the function is updated with a new Tk value.

Next we enter a large while loop, that runs so long as the desired number of sampled points, 'n', has yet to be accepted. The z value is computed, and a random sample point from the function is drawn, as chosen by the lower level function 'rs', the inverse sampling function.. If that point falls outside the current bounds of the Tk points, that point is added and the lower level chords (and their respective coefficients) are updated with 'lupdater'. Next, another lower level function, a binary search engine, is called to find which chord corresponds to the chosen sample point. The coefficients are applied to calculate the lower bound value, and the function 'du.unnormalized' is called to compute the upper bound.

That lower level function, 'du.unnormalized', is a workhorse that takes as inputs the sampled point, the current abscissae, the z value and the lower and upper bounds and computes the value of the upper envelope. This involves another sub-function which computes the normalization constant, 'c', and integrates over the upper envelope.

Once both the lower and upper bound are computed, we grab a random number from the uniform distribution, as is part of the ARS process. If the uniform value is less than than the difference between the exponentials of the log value and the upper envelope value (aka the squeezing test), the point is immediately accepted and the TK abscissae updated. If not, then we check that the uniform point is less than the difference between the exponents of the lower and upper function values. If this is true, then the point is accepted, else it is rejected. We only evaluate the log of that point if we fail to squeeze, as it is computationally expensive.

Once we've reached the desired number of sampled points, the while loop ceases, the function is halted and the vector of final values is returned.

Group Member Responsibilities

Weixin Cai served as the chief architect of the algorithm and the leader of the project. Meeting with members individually and together, Weixin broke down the process into four critical sections and helped to make sure each group member was both involved in and informed of each section while maintaining leadership over one section in particular. He also served as the chief manager of the GitHub repository and was the go-to member for coding questions and general outline.

Weixin himself also wrote the invaluable and computationally difficult underlying code that computed the upper envelope piecewise function, all in vectorized fashion. This involved subfunctions that: a) created an abscissae of 'k' starting Tk points, b) integrated over the exponentiated upper envelope, and c) computed the inverse sampling of the upper envelope density. He also offered guidance on vectorizing the rest of the code, testing, formulating checks, warnings and error messages, and building the R package.

HR Huber-Rodriguez was chiefly in charge of the second level of the function; the creation of the linear lower bound piecewise function. This was done by creating a matrix of slope and y-intercept coefficients for all chords associated with Tk values, running a binary search on each sampled point to find the correct chord, and drawing new chords from the updated Tk points. He was also responsible for implementing the squeezing and rejection steps of the rejection sampling process and for returning the ultimate vector of sampled points. HR also tested the code on a variety of standard distributions by using R's built-in density functions of these distributions and comparing the similarity of the results to a random generation from the same distribution using the KS test.

Jianbo Chen was responsible for writing the updating step of the function, which involved locating the index of the desired point, calculating the associated coefficients for the upper and lower functions, and updating those functions directly. Because adaptive rejection sampling is often used when a density function is computationally expensive to evaluate, Jianbo proposed testing the code by evaluating a kernel density function, which can be found in the package manual. He generated graphs and plots that more clearly demonstrate the process of adaptive rejection sampling, found at the end of this report. Jianbo also offered suggestions on the vectorization of the code as well as ways to increase speed and efficiency.

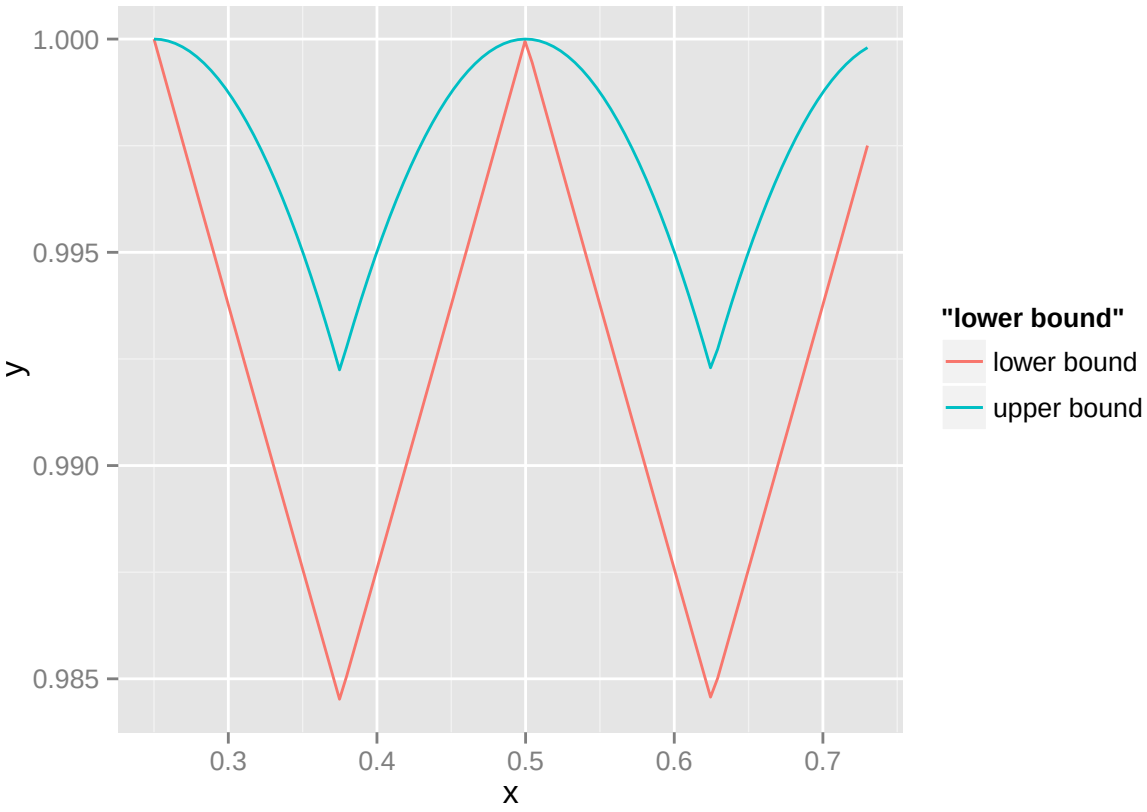
Chenyu Wang was chiefly responsible for compiling all the sections of code into one compact R package, and wrote the entire manual. She was responsible for writing the accompanying documentation and implemented the testing function within the package. She also designed the crucial check for log concavity, which is paramount to correct usage of the algorithm, by ensuing that the derivative of the left end-point was greater than the derivative of the right end-point.

As a whole, the project came together piece by piece, with group members often offering input on one another's sections to ensure familiarity with the goal of the project as well as with the idiosyncrasies of individual sections of code, which would require tidy sutures in order to weave together one coherent algorithm and one effective R package.

Plot

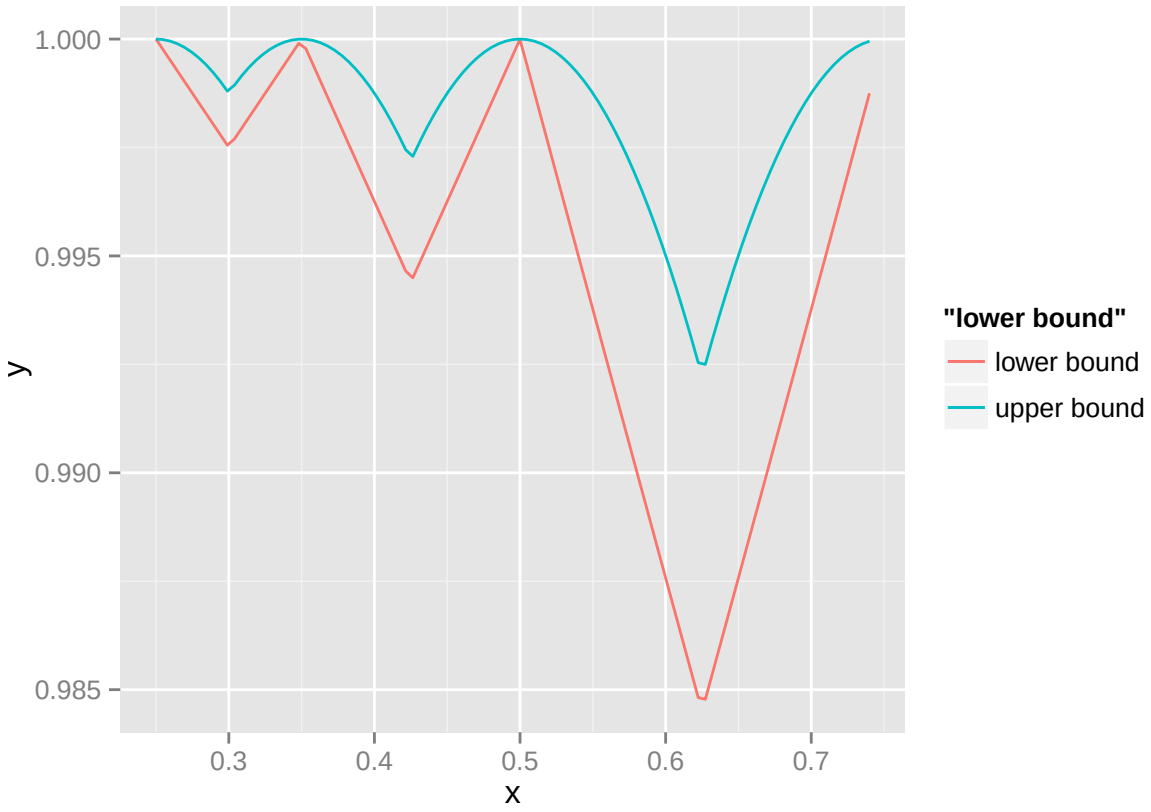
The following plot shows the upper and lower bound functions for a normal distribution, with k = 5.

```
## Warning in if (index >= length(Tk) | index < 1) {: the condition has length
## > 1 and only the first element will be used
```



And one more plot after updating once, thus adding a single Tk point.

```
## Warning in if (index >= length(Tk) | index < 1) {: the condition has length
## > 1 and only the first element will be used
```


1.000
0.995
0.990
0.985
y
0.3
0.4
0.5
0.6
0.7
x
"lower bound"
lower bound
upper bound