STAT 243 Final Project: Adaptive Rejection Sampling

Jinxuan Fan, Andrea Padilla, Matthew Song

Due: 12/15/2022

Introduction

The Github Repository for our project is located at: `https://github.berkeley.edu/mgsong/ars`

This Final Project seeks to implement the Adaptive Rejection Sampling (ARS) algorithm outlined by Gilks and Wild (1992) in their seminal paper "Adaptive Rejection Sampling for Gibbs Sampling".

Briefly, the ARS algorithm works for log-concave distributions. It considers an upper-hull envelope which constrains the (log-)function from above, similar to normal Rejection Sampling. It also constrains the function from below using a lower envelope. Crucially, the adaptive rejection sampling algorithm samples points and updates the points to create the upper-hull envelope, which makes the rejection sampling more and more precise and decreases the rate of rejections, thus increasing efficiency. Besides the original paper by Gilks and Wild (1992), we found this link: https://lucius-yu.github.io/docs/probability/BasicMCSamplingMethod/ to be a helpful resource, particular in the sampling step of the algorithm (note that the contents of this link are written in Chinese, though Google Translate has a decent translation).

For testing, we graphically show that the function approximates various distributions well in a later section. Numerically, we decided to look at the mean and variance of our sample output versus the closed-form mean and variance of the true underlying distribution, and calculate confidence intervals to show that the values are not significantly different. **Reminder note for Chris: for some reason, the command `test_packages("ars")` cannot find the test files in the package, but running it directly through devtools or directly referencing the file works properly (Andrea talked to you about this issue).**

In terms of contributions from group members:

- Jinxuan wrote the `abscissae()` function and worked on putting together the packaging, testing, and documentation.
- Andrea drafted the writeup and worked on putting together the packaging, testing, and documentation.
- Matthew coded up the remainder of the functions, drafted the writeup plus appendix, identified test cases to be implemented in the testing package, and wrote documentation for the functions.

Overview of Functions

Main Function

ars <- function(g, n, absc = c(), lower = -Inf, upper = Inf)

This is the main function of this project. It takes in five parameters and returns a vector of samples generated using the Adaptive Rejection Sampling algorithm. For the inputs:

- `g` is a necessary input. It is the target distribution that the user wishes to sample from. `g` must be log-concave (the function has checks in case of non-log-concavity, and will alert the user if it detects that), but can be unnormalized. Note that `g` gets converted to `h = log(g)` by the function `eval_h` (described below). Hereafter, we will refer to the function `h` instead of `g` where appropriate.

- `n` is an optional input. It is the size of the sample which the user wants to generate, and is 5000 by default (so that the test cases shown below can better approximate the true distribution).
- `absc` is an optional input. It is the vector of initial abscissae points, and is empty by default. If the user does not supply a vector of abscissae points, the function will attempt to choose proper initial abscissae points according to `g`. If the user supplies only one abscissae point, the function will attempt to find a second abscissae point that satisfies initial conditions (described more in detail in the `abscissae()` function section). If the user supplies two or more initial points, the function assumes that they are legitimate and will check whether it can use those abscissae points to run the algorithm. If not, it will alert the user and exit (instead of replacing the bad input with useable initial abscissae points).
- `lower` is an optional input. It is the lower bound of the distribution, and is set to -Infinity by default.
- `upper` is an optional input. It is the upper bound of the distribution, and is set to +Infinity by default.

At a high level overview, the `ars` function has the following order of operations:

- Step 0: Sanitize user inputs (check validity of user inputs)
- Step 1: Run the initialization
  - Confirm user-input abscissae points are valid, or generate a set of valid abscissae points.
  - Calculate relevant values for use in the ARS algorithm
- Step 2: Sample points, updating the initial set of abscissae as necessary
- Step 3: Return the vector of sampled points when complete.

Helper Functions

sanitize_inputs(g, n, a, l, u)

This is a function to check that the user inputs pass basic checks. `g`, `n`, `a(bsc)`, `l(ower)`, `u(pper)` are the inputs from the main ars() function. We checked that:

- `g` is a function
- `n` is a positive integer
- `absc` is a numeric vector, or empty
- `lower` and `upper` are single-valued numerics (note that -Inf and Inf are numeric), with `upper` > `lower`
- The values in `absc`, if provided by user, are within the range from (`lower`, `upper`)

initialize_ars(a)

This is a primary function which initializes values used in the ARS algorithm. It takes in the vector of abscissae points `a` provided from the initial `ars` call (so it can be empty).

There are three sub-functions:

- `log_concavity(hd)`, which takes in a list of derivative values evaluated from the `h` function and checks whether `g` is log-concave by verifying that the derivative values are non-increasing.
- `removeDupe(vec)`, which takes in a vector and flags the second element of each pair of successive values if it is a duplicate. This is used for efficiency when applied to the list of abscissae points or derivative values.
- `valid_abscissae(hd)`, which takes in a vector of derivative values and checks whether or not the current set of abscissae points satsisfies initial conditions if the function is unbounded from above or below. In particular, according to Section 2.2.1 of the Gilks and Wild paper:
  - If a lower bound is not finite, then we must have the initial (lowest-value) abscissae point x_1 with $h'(x_1) > 0$, where $h(x) = log(g(x))$ and g is the user-input function.
  - If an upper bound is not finite, then we must have the final (highest-value) abscissae point x_k with $h'(x_k) < 0$
  - Note that this implies that if the function is not bounded from above or below, then we must have two initial abscissae points (as a singular point cannot have derivative that is simultaneously > 0 and < 0).

For the main part of `initialize_ars(a)`, we first create a vector of two valid abscissae points if two are not supplied by the user. We order the list of abscissae points (used for comparison throughout the code), and create all of the necessary values outlined by the Gilks and Wild paper, running the above three sub-function checks to ensure we have initialized our values properly.

The function returns cleaned versions of the list of abscissae points `absc`, the log value of the density `h_vals` (evaluated at the abscissae points), the derivative of the log-density `hderiv_vals` (evaluated at the abscissae points), and the tangent-intersect points `z_vals`.

abscissae(a)

This function takes in the list of abscissae points provided in the original `ars` call, as the `abscissae(a)` function gets evaluated immediately. It generates valid initial abscissae points, if none are provided. By utilizing a grid search with step size 0.01, the function attempts to find the value where the gradient is approximately 0, and then selects a point to the left and right of that point where the gradient is negative and positive (see discussion in `initialize_ars` section). Moreover, if the user only provides one point, we first check the relationship with the 0-gradient point we find. For example, if the user-provided point is larger than the 0-gradient point, the function finds one point on the left of the 0-gradient point to be the other initial abscissae point.

Note that the bounds for this search are for x-values between -50 and 50, so if the distribution is ill-defined on this range, the search will fail.

eval_h(x)

This function takes in a vector and evaluates the the function `h` at the point `x`. There is also a check that the evaluation does not return NaN, which would imply that the original `g` is negative at `x`. If this happens, the function will alert the user and ask them to fix their density.

eval_z(a, h ,hd)

This function takes in the abscissae vector `a`, a vector of `h` values evaluated at the abscissae points, and the vector of `hd` values of the derivative of the `h` values.

By using the equation provided in the article, which is $z_j = \frac{h(x_{j+1}) - h(x_j) - x_{j+1}h'(x_{j+1}) + x_j h'(x_j)}{h'(x_j) - h'(x_{j+1})}$, we get a vector of the tangent points which are the intersection points of the tangent lines at the abscissae as output.

eval_u(x, z, a, h, hd)

This function takes in a vector `x` to evaluate at, the vector `z` of tangent point values, `a` of abscissae points, `h` of h-values, and `hd` of derivatives. It then calculates the corresponding upper hull point value at `x`, according to the Gilks and Wild paper formula. This is piecewise, so there is code to determine what range of the piecewise upper-hull a specific `x` point lands in, and then calculates its corresponding value as output.

eval_l(x, a, h)

This function takes in a vector `x` to evaluate at, `a` of abscissae points, and `h` of h-values. It then calculates the lower hull/squeezing function, according to the Gilks and Wild paper formula. It returns the lower-hull value.

eval_sample(a, z, h, hd)

This function takes in values with the same definitions as previous functions. It draws samples from our target function according to the ARS algorithm (see appendix for writeup). The first step is to proportionally select the upper hull piece that should be sampled, and then the second step is to actually sample a point within that upper hull piece range.

accept_reject(x, z, a, h, hd)

This is a very important function in the algorithm. With quite a lot of the ground work being done in the functions described above. It takes in a vector `x` to evaluate at, and the remaining inputs are analogous to previous functions.

It determines whether to accept a sample value `x`, and whether to add it to the abscissae pool if accepted. As explained in the paper by Gilks and Wild, there are two steps to this: the squeeze test, and the rejection test. If the point lies below the lower hull then we automatically accept it because then it is certainly below the target function itself. By doing this we don't need to evaluate $h(x)$ as often. If it is not below the lower hull then we can move on to the rejection step, and determine whether the point lies under the target function. If so, then we add the sample to the list of abscissae points and re-initialize (using `initialize_ars`) our list of points for the algorithm. The output is a value that determines whether we reject, accept and do nothing, or accept and update.

Note that there is an edge case to consider. Assume we test $x \in z_i, z_{i+1}$. If $eval_l = -Inf$, then $exp(eval_l - eval_u) = 0$ (i.e. the function will always run the second comparison check). More importantly, if $eval_h = eval_u$ (which happens when $h' = 0$, by definition of $u(x)$), then $exp(eval_h - eval_u) = 1$, and the function will always accept (and since this is the second check, it will update the abscissae). But this is inefficient, since the updated abscissae point will still be constant at the new point! Furthermore, the removal of the extraneous (second) abscissae point means that the function will only accept "more extreme" abscissae i.e. closer to the bounds, which may run into numerical precision issues/undefined derivatives. Thus, the function is specifically instructed to not update the abscissae in this special circumstance.

Main Loop

In the main loop, the function initializes some counter variables and set a tolerance threshold for comparing values (i.e. testing if two values are equal, used in `removeDupe` under the `iniitalize_ars` function) to be `1e-8`, which is empirically determined to be an acceptable threshold (see Appendix).

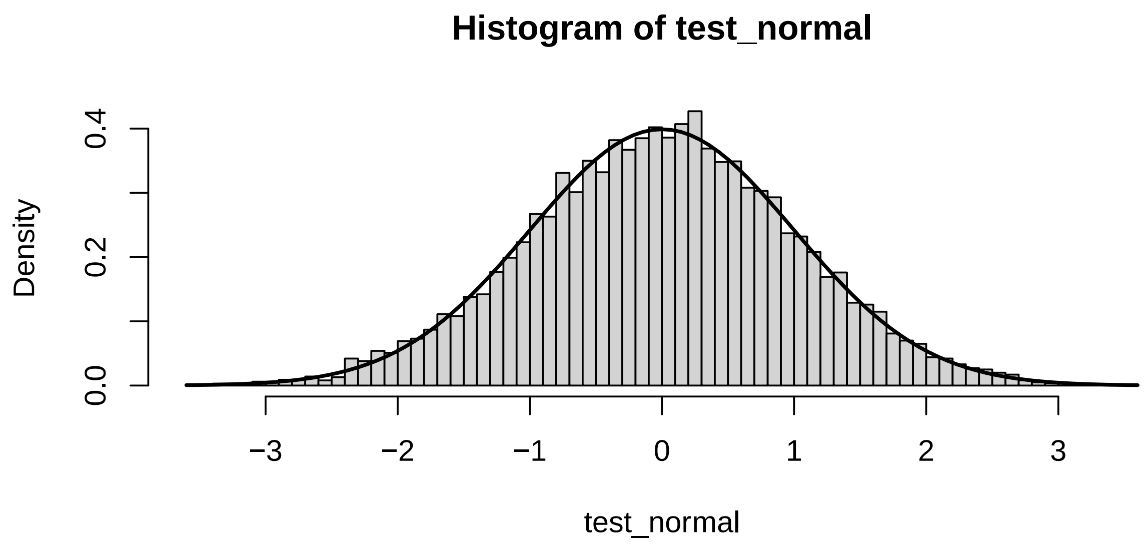
Then, the function iterates by sampling points according to the ARS algorithm until either the number of required samples is satisfied, or too many runs have passed (default set to be 10 times `n`).

Once the sampling is done, the function returns the vector of sampled points.

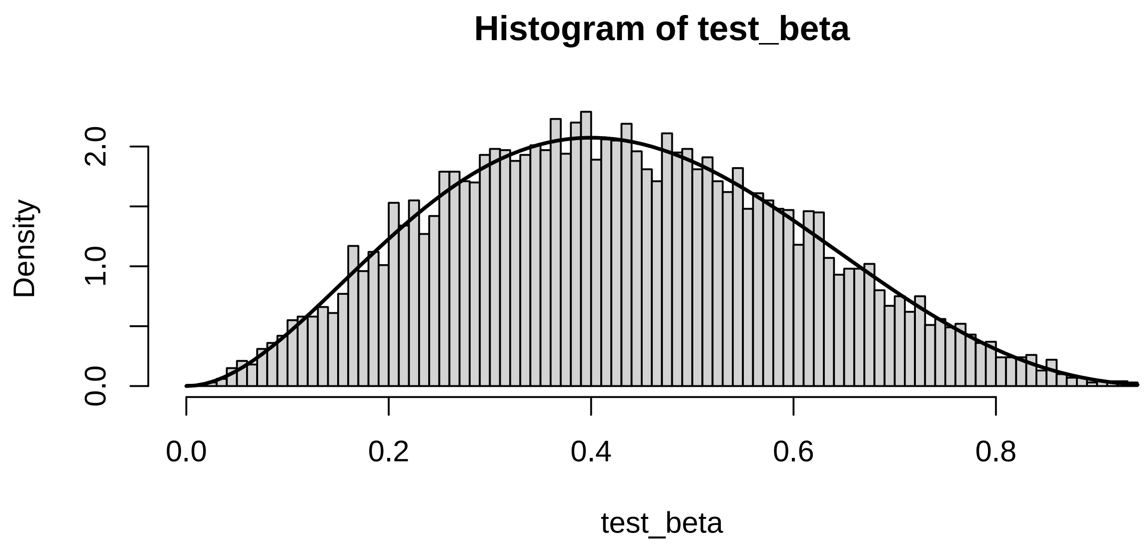
Example Usage Cases

We provide a few examples of the `ars` sampling output versus the true underlying distribution. More tests for other functions are found in the `testthat` file within the package. Note that the exponential distribution has constant slope (after taking the log), which requires specific attention (see appendix for sampling writeup addressing this case).

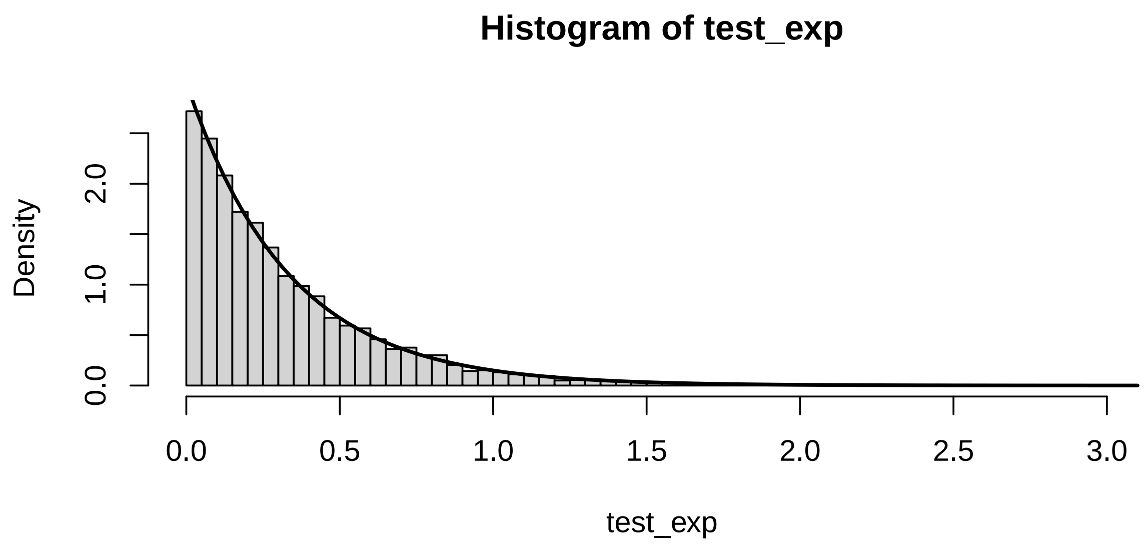
```r
library(ars) # Assumes the `ars` package is already installed properly
library(assertthat)
library(testthat)
library(numDeriv)
set.seed(322)
n <- 10000 #for better sample histogram distribution

# Normal(0,1)
test_normal <- ars(dnorm, n)
hist(test_normal, breaks = 100, freq = FALSE)
curve(dnorm(x), lwd = 2, add = TRUE)
```



```r
# Beta(3,4)
beta_func <- function(x) {dbeta(x, 3, 4)}
test_beta <- ars(beta_func, n, absc = c(0.2, 0.6), lower = 0, upper = 1)
hist(test_beta, breaks = 100, freq = FALSE)
curve(dbeta(x, 3, 4), lwd = 2, add = TRUE)
```

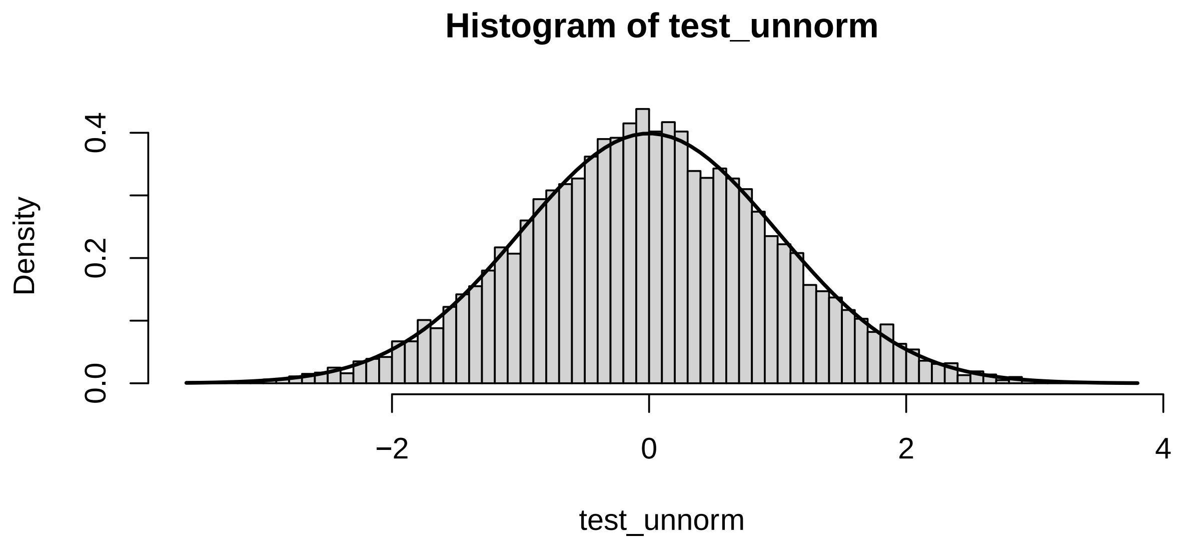


```r
# Exp(3)
exp_func <- function(x) {dexp(x, rate = 3)}
test_exp <- ars(exp_func, n, absc = c(1, 4), lower = 0, upper = Inf)
hist(test_exp, breaks = 100, freq = FALSE)
curve(exp_func(x), lwd = 2, add = TRUE)
```



The function is also robust to non-normalized densities.

```
#takes out the 1/sqrt(2pi) normalizing constant for standard normal
unnorm_normal <- function(x) {sqrt(2*pi)*dnorm(x)}
test_unnorm <- ars(unnorm_normal, n)
hist(test_unnorm, breaks = 100, freq = FALSE)
#use the standard normal density as comparison, fits well
curve(dnorm(x), lwd = 2, add = TRUE)
```



The function is able to identify non-log-concave input functions (no output below since `expect_error` succeeds).

```
cauchy_func <- function(x) {dcauchy(x, location = 2, scale = 4)}
#This should fail, Cauchy is non-log-concave.
```

```r
expect_error(ars(cauchy_func, n, absc = c(-3, 3), lower = -Inf, upper = Inf))
```

And the function is able to check whether initial abscissae points are provided properly, when the user provides 2 or more points (no output below since `expect_error` succeeds in each case).

```r
# Logistic(1, 2) with Abscissae = c(2, 3).
# Both initial abscissae points have negative slope, so it fails because unbounded from below.
logis_func <- function(x) {dlogis(x, location = 1, scale = 2)}
expect_error(ars(logis_func, n, absc = c(2, 3)))
expect_error(ars(logis_func, n, absc = c(2, 3), upper = 10))
#unbounded from above, not an issue
expect_error(ars(logis_func, n, absc = c(2, 3), lower = -10), NA)
```

Appendix

Sampling Algorithm for x^* in ARS

Notation:

- $h(x) = \log(g(x))$ where $g(x)$ is the (possibly unnormalized) user-input density.
  - $h'(x) = \frac{d}{dx}h(x)$ as usual
- $a_1, a_2, ..., a_k$ are the (ordered) Abscissae Points (k total)
- $z_0, z_1, ..., z_k$ are the (ordered) Tangent Intersect Points ($k+1$ total. Initially, set $z_0 = -\infty$ and $z_k = \infty$)
- $u_i(x) = h(a_i) + (x - a_i)h'(a_i)$, this is the upper hull value at a point $x \in [z_{i-1}, z_i]$ (i.e. depending on the value of x, it lies between z_{i-1} and z_i, so i depends on the choice of x)

Step 1: Sample Segment $[z_{i-1}, z_i]$

Idea: Want to get the area under the curve of $exp(u_i(x))$ for each piecewise part $u_i(x)$, then sum up areas and standardize to get a CDF. Then, sample $p_1 \in Unif(0,1)$ and find which index i that p_1 falls into, i.e. calculate i such that $p_1 \in [z_{i-1}, z_i]$ (comparing p_1 against the standardized CDF values for each segment $[z_{i-1}, z_i]$).

Assume we want to find the area under the curve of $exp(u_i(x))$ for $x \in [z_{i-1}, z_i]$. That is, we want to calculate:

$$\int_{z_{i-1}}^{z_i} exp(u_i(z))dz$$

Note that $u_i(x) = h(a_i) + (x - a_i)h'(a_i) \implies exp(u_i(x)) = exp(h(a_i) + (x - a_i)h'(a_i))$. Then:

$$\begin{aligned}
\int exp(u_i(z))dz &= \int exp(h(a_i) + (z - a_i)h'(a_i))dz \\
&= \frac{1}{h'(a_i)} exp(h(a_i) + (z - a_i)h'(a_i)) \\
&= \frac{1}{h'(a_i)} exp(u_i(z)) \\
\implies \int_{z_{i-1}}^{z_i} exp(u_i(z))dz &= \frac{1}{h'(a_i)}(exp(u_i(z_i)) - exp(u_i(z_{i-1})))
\end{aligned}$$

Thus, we just have to calculate the last quantity to get the area under the curve for each $[z_{i-1}, z_i]$. Do this, then standardize so that probabilities are between $[0, 1]$, and then sample a random p_1 and calculate to which i that p_1 corresponds to.

Step 2: Sample x^* within Segment $[z_{i-1}, z_i]$

Following the above algorithm, we know which segment we want to sample from: $[z_{i-1}, z_i]$. The idea now is to again use inverse CDF on this segment. Assume we draw $p_2 \in Unif(0, 1)$. To calculate the CDF of $exp(u_i(x))$ for $x \in [z_{i-1}, z_i]$, I note that:

$$CDF = P(X \leq x) = \frac{\int_{z_{i-1}}^{x} exp(u_i(z))dz}{\int_{z_{i-1}}^{z_i} exp(u_i(z))dz}$$

where the denominator is a constant, and is simply the area under the curve for that specific segment, which we can calculate (or already have calculated) from the derivation in Step 1. Let $\int_{z_{i-1}}^{z_i} exp(u_i(z))dz = C$ for notation. Then we can solve for x^* as follows, to get the inverse CDF:

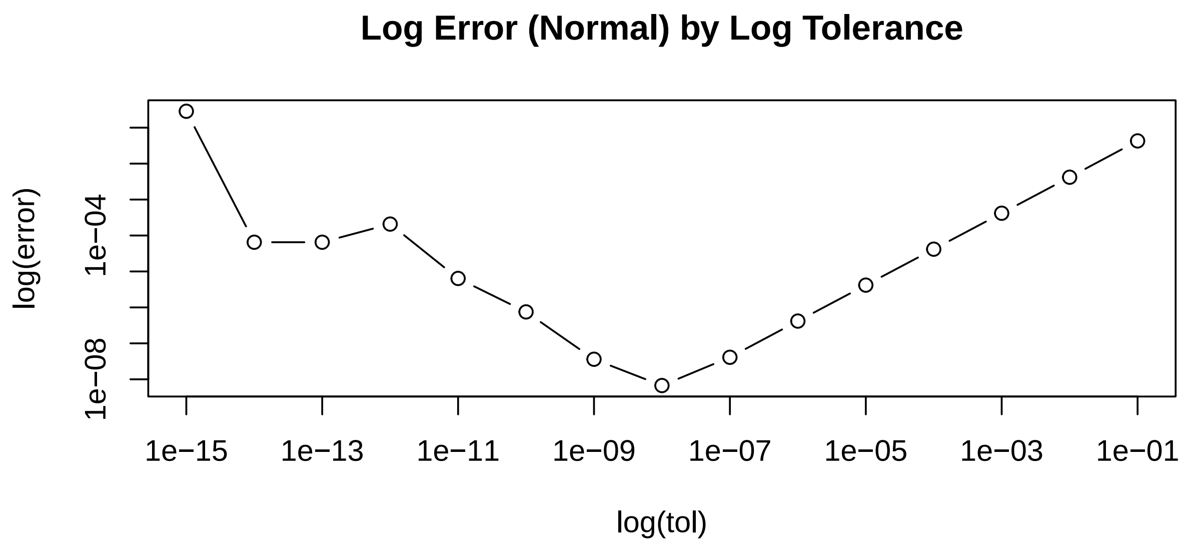
$$
\begin{aligned}
p_2 &= \frac{\int_{z_{i-1}}^{x} exp(u_i(z))dz}{C} \\
p_2 C &= \int_{z_{i-1}}^{x} exp(u_i(z))dz \\
&= \frac{1}{h'(a_i)}(exp(u_i(x^*)) - exp(u_i(z_{i-1}))) \\
h'(a_i)p_2C + exp(u_i(z_{i-1})) &= exp(u_i(x^*)) \\
\log(h'(a_i)p_2C + exp(u_i(z_{i-1}))) &= u_i(x^*) \\
&= h(a_i) + (x^* - a_i)h'(a_i) \\
\frac{-h(a_i) + \log(h'(a_i)p_2C + exp(u_i(z_{i-1})))}{h'(a_i)} + a_i &= x^*
\end{aligned}
$$

Thus, after sampling p_2 from $Unif(0,1)$, we plug into the above formula to derive our desired x^*.

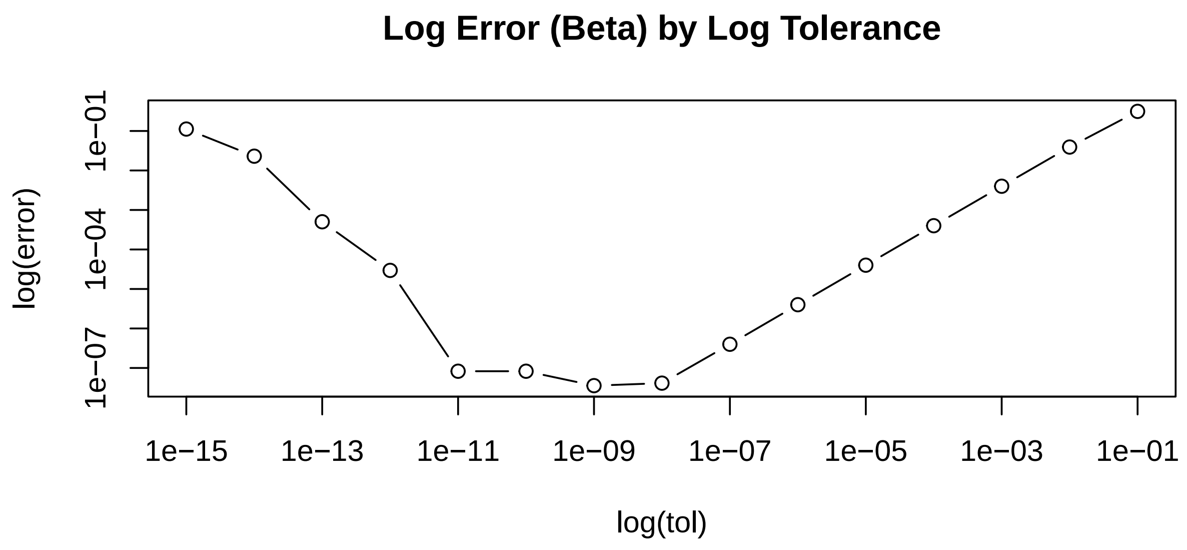
Empirical Tolerance Testing

Based on ideas from Quiz 2, Question 3, we empirically run tests to determine a reasonable threshold for tolerance. The lecture notes mention that $sqrt(\epsilon) \approx 1e-8$ (where ϵ is machine epsilon) is a good threshold. We select a few distributions with known closed-form derivatives, and evaluate the derivative at a known point for various values of potential tolerance values and plot the relative error (on the log-scale).

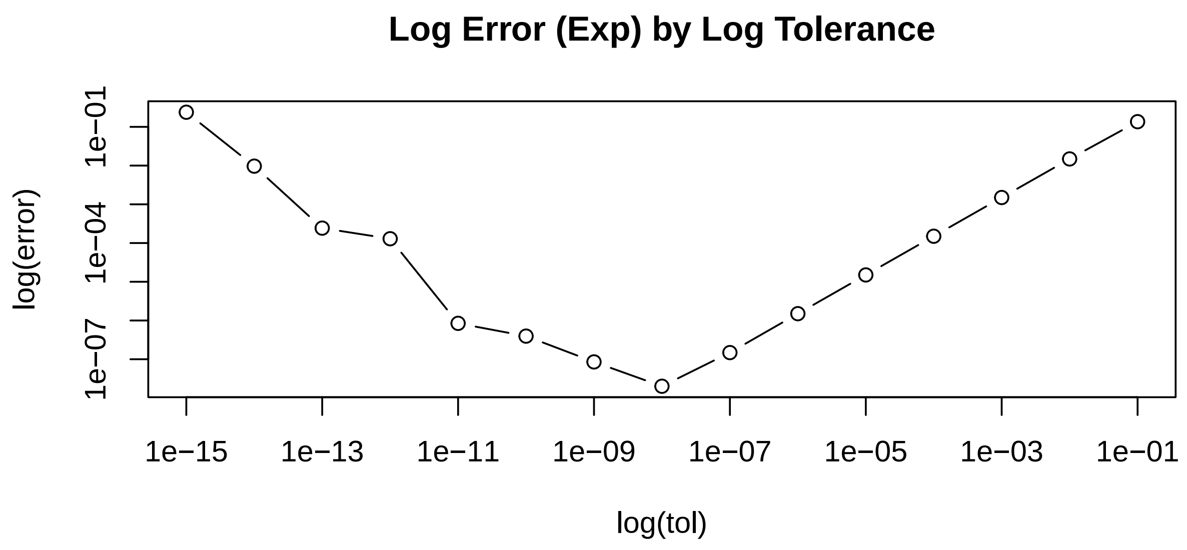
```r
tol_seq <- 10^(seq(-1, -15, by = -1)) #1e-1 to 1e-15

# Standard Normal, evaluated at x = 1.5
manual_deriv_norm <- function(eps) {
  return((dnorm(1.5 + eps) - dnorm(1.5))/eps)
}
target_norm <- -1.5 * exp((-1.5^2)/2)/sqrt(2*pi) #Closed-Form Derivative of N(0,1)
error_norm <- abs((manual_deriv_norm(tol_seq) - target_norm)/target_norm)
plot(tol_seq, error_norm, log = 'xy', type = "b",
     xlab = "log(tol)", ylab = "log(error)", main = "Log Error (Normal) by Log Tolerance")
```



```r
# Seems like error is minimized at tol = 1e-8

# Beta(3,4), evaluated at x = 0.5
manual_deriv_beta <- function(eps) {
  return((dbeta(0.5 + eps, 3, 4) - dbeta(0.5, 3, 4))/eps)
}
target_beta <- (-0.5 * (5*0.5 - 2) * (1-0.5)^2) * 1/beta(3, 4) #Closed-Form Derivative of Beta(3,4)
error_beta <- abs((manual_deriv_beta(tol_seq) - target_beta)/target_beta)
plot(tol_seq, error_beta, log = 'xy', type = "b",
     xlab = "log(tol)", ylab = "log(error)", main = "Log Error (Beta) by Log Tolerance")
```



```r
# Seems like error is minimized at tol = 1e-8 or 1e-9
# Exp(3), evaluated at x = 1
manual_deriv_exp <- function(eps) {
  return((dexp(1 + eps, rate = 3) - dexp(1, rate = 3))/eps)
}
target_exp <- -9*exp(-3*1) #Closed-Form Derivative of Exp(3)
error_exp <- abs((manual_deriv_exp(tol_seq) - target_exp)/target_exp)
plot(tol_seq, error_exp, log = 'xy', type = "b",
     xlab = "log(tol)", ylab = "log(error)", main = "Log Error (Exp) by Log Tolerance")
```



```r
# Seems like error is minimized at tol = 1e-8
```

We can see that the error is typically minimized at $1e - 8$ tolerance, as expected from the discussion above.