Adaptive rejection sampling algorithm project

Lauren Ponisio, Katherine Ullman, Xinyue Zhou and Cenzhuo Yao

December 16, 2015

Github repository for the final project is under username z35712526

1 Approach

We have two main objects that we create and update in our function. The first is the *info* matrix which stores the proposed sample (x), the function evaluated at x, and the derivative of the function evaluated at x. The second is *ret* which stores accepted samples values and is returned.

We begin by creating our empty *info* matrix using the approximation for the number of iterations from Gilks et al. 1992. We then fill it with initial values supplied by the user or by values or the defaults of -inf to inf.

```
## function for initialize the info matrix in unbounded cases
## aa and bb are the bounds, info is the info matrix with x, fx, and
## f'x, FUN is the function of interest and DD is the derivative
## function
## the function returns the updated info matrix
help_init_info <- function (aa, bb, info, FUN, DD){
  info[1, 1] <- aa
  info[1, 2] <- FUN(info[1, 1])
  if(!is.finite(info[1, 2])) {
    stop('aa is not defined on f', call. = FALSE)
  }
  info[1, 3] <- DD(info[1, 1])
  info[2, 1] <- bb
  info[2, 2] <- FUN(info[2, 1])
  if(!is.finite(info[1, 2])) {
    stop('bb is not defined on f', call. = FALSE)
  }
  info [2, 3] <- DD(bb)
  return(info)
}
```

We then set our counters and enter a while loop.

Within the while loop we first check to see if the function is log-concave using the *check_concave* function. That function checks if the derivative of the function of interest evaluated at the proposed samples is always decreasing.

```
## checks log-convexity
## clean_info is the ordered info matrix
check_concave <- function(clean_info){
  if(is.null(clean_info[,3])) {return(TRUE)}
  return(prod(round(clean_info[,3][-1],5) <=
```

```
              round(clean_info[,3][-nrow(clean_info)], 5)))
}
```

Next, a new proposed sample is taken (x) using *sample_envelope*. The function draws two random uniform numbers. The first draw is to determine which piece of the upper bound to sample from. The second draw is to calculate the sample x by evaluating the inverse cdf at the value of the random draw. The function returns the proposed sample.

```
## returns a new sample given the info matrix and z
## input is clean info matrix and clean z
sample_envelope <- function(samp_info, samp_z, num_sample){
  ## segments of bounds
  p <- rep(NA, nrow(samp_info))
  p <- exp(samp_info[,2])*(exp((samp_z[-1] - samp_info[,1])*
                               samp_info[,3]) -
                           exp((samp_z[-nrow(samp_info) - 1] -
                                samp_info[,1])*
                               samp_info[,3]))/samp_info[,3]
  ## q for normalizing p
  q <- p/sum(p)
  ## which region we sample from
  w <- runif(num_sample)
  i <- vector(mode = 'numeric', length = num_sample)
  for (j in 1:num_sample){
    i[j] <- sum(cumsum(q) < w[j]) + 1
  }
  ## sample pi using inv cdf
  samp_x <- (log(p[i]*runif(num_sample)*samp_info[i, 3] +
                   exp(samp_info[i,2] +
                         (samp_z[i] - samp_info[i,1])*samp_info[i,3])) -
               samp_info[i,2])/samp_info[i,3] + samp_info[i,1]
  return(samp_x)
}
```

We then check if it is defined on the function of interest and draw a new x if it is not. We also check whether x has already been drawn and save it in the output if it has been.

We then draw a w from the uniform(0,1) that will be used to identify whether the sample is accepted or rejected.

We next calculate the lower bound using the function *line_fun* which takes two points and calculates the slope between those points, and evaluations that line function to return the y value.

```
## takes two points and returns the function of that line
line_fun <- function(x1, y1, x2, y2, x_star){
  if(x1 == x2) {stop('Two points in a vertical line')}
  y_star <- (y2 - y1)/(x2 - x1)*(x_star - x1) + y1
  return(y_star)
}
```

For the upper bound, we only have one point and the derivative, so we used the function *line_fun_p* to evaluate the line at that point.

We then determine whether our sample:

1. falls within the lower bound. If it does, we accept it and store it in *ret*

2. falls between the lower bound and the function of interest. If it does, we accept it and store it in ret. We also evaluate the function and its derivative at the sample and update the $info$ matrix using $update\_info$ function.

3. falls between the function and the upper bound. If it does, we reject the sample. We also again update the $info$ matrix.

We continue in the while loop until the desired number of accepted samples is reached.

1.1 Special cases

For the uniform distribution, the derivative is always zero. If for the bounded uniform, the derivative at both of the provided initial condition is zero, it must be the uniform so we just draw using $runif$.

1.2 Numerical differentiation

We created a function for numerical differentiation.

We choose ϵ to be 1^{-8} and then evaluated:

$$f' \propto (f(x+\epsilon) + f(x-\epsilon))/2\epsilon \tag{1}$$

Or at the boundaries:

$$f' \propto (f(x) + f(x-\epsilon))/\epsilon \tag{2}$$

$$f' \propto (f(x+\epsilon) + f(x))/\epsilon \tag{3}$$

```
## function for numerically approximating the derivative
## x is the point to evaluate the derivate at, FUN is the function,
## a and b are the bounds
## the function returns the approximation to the derivative at the point of
## evaluation
Derv <- function(x, FUN, a, b){
  if (x == a) {return ((FUN(x + 1e-8)-FUN(x))/1e-8)}
  if (x == b) {return ((FUN(x) - FUN(x - 1e-8))/1e-8)}
  if (a <= x && x <= b) {return((FUN(x + 1e-8)-FUN(x - 1e-8))/2e-8)}
}
```

1.3 Efficiency

1.3.1 Vectorization

All of the algorithm is vectorized.

1.3.2 Sorting

After each update, we needed to re-sort the $info$ matrix and z vector. Instead of using the build-in $sort()$ function, we just found the index where the new value needed to be added.

1.3.3 Sampling in batches

After originally sampling, one value at a time, we extended the function to sample batches of proposed samples at a time.

We had to decide when it was most efficient to take one sample at a time and update the bounds, or draw multiple samples. We decided to first look at the *hit_rate*, which is the number of accepted samples out of the number of total draws. When $hit\_rate = 1$, this means that we have not updated our bounds. This is because we are early in our iterations and do not have a good approximation of our function. So, if $hit\_rate = 1$, we draw one sample at a time. If the *hit_rate* is close to 1, it means that most of the samples we draw will go to sample and thus our approximation of the function with the upper bound and lower bound is good. If $hit\_rate < 1$, we draw $1/(1 - hit\_rate)$ sample at a time.

2 Testing

2.1 Final tests

To test our algorithm we compared our samples to known distributions using Kolmogorov-Smirnov test. We tested our function against known log-concave distributions including the normal, gamma, Laplace, logistic, beta and uniform (Fig. 1).

Our algorithm consistently passes for all of the functions. Sometimes the samples did fail however, likely because the numerical differentiation does not preform well.

2.2 Defensive programming

We created to checks to determine whether the inputs arguments were in the expected format. We also checked if the derivative function provide by the user was the correct derivative function.

We also created a function that checks whether the input function is log-concave by checking whether the derivative is always decreasing. We added checks for when the function was not defined on the sample.

2.3 Unit tests

We used Browser() to do informal testing of each module. Because our modules were all uncomplicated (most sort objects and do basic arithmetic), we could not think of sensible formal tests to perform for most of the modules. We performed a unit test for the most complex function, *sample_envelope* using the exponential.

Because the exponential distribution is fairly simple, we could create an $info$ matrix and z vector that encompassed the entire distribution. Therefore we could compare the samples from *sample_envelope* to the theoretical exponential using a Kolmogorov-Smirnov test.

```
sample_envelope <- function(samp_info, samp_z, num_sample){
  ## segments of bounds
  p <- rep(NA, nrow(samp_info))
  p <- exp(samp_info[,2])*(exp((samp_z[-1] - samp_info[,1])*
                               samp_info[,3]) -
                           exp((samp_z[-nrow(samp_info) - 1] -
                                samp_info[,1])*
                               samp_info[,3]))/samp_info[,3]
  ## q for normalizing p
  q <- p/sum(p)
  ## which region we sample from
  w <- runif(num_sample)
  i <- vector(mode = 'numeric', length = num_sample)
  for (j in 1:num_sample){
    i[j] <- sum(cumsum(q) < w[j]) + 1
```

```
  }
  ## sample pi using inv cdf
  samp_x <- (log(p[i]*runif(num_sample)*samp_info[i, 3] +
                   exp(samp_info[i,2] +
                         (samp_z[i] - samp_info[i,1])*samp_info[i,3])) -
               samp_info[i,2])/samp_info[i,3] + samp_info[i,1]
  return(samp_x)
}


## test using the exponential
set.seed(4)
a <- 0
b <- 1
lambda <- 1

samp_info <- matrix(c(a, b,
                      log(dexp(a, rate=lambda)),
                      log(dexp(b, rate= lambda)),
                      -lambda,
                      -lambda), nrow=2)
samp_z <- c(0, 0.5, Inf)

test.out <- sample_envelope(samp_info, samp_z, 10000)

out.test <- ks.test(test.out, pexp)

if(out.test$p.value > 0.05){
  print("sample matches exponential, sample_envelope unit test passed")
}else{
  print("sample does not match exponential, sample_envelope unit test failed")
}

## [1] "sample matches exponential, sample_envelope unit test passed"
```

The real test is whether our overall function was able to reproduce samples from known distribution, which was the case.

3 Contributions

Zhou and Yao wrote the skeleton of the algorithm, and all members contributed to de-bugging, improving efficiency, and improving style. Zhou and Yao finalized the functions. Lauren and Katherine wrote the testing functions, and wrote the first draft of the write up. All members contributed to revisions. Katherine wrote the help documentation. We worked together to package the code into a package.

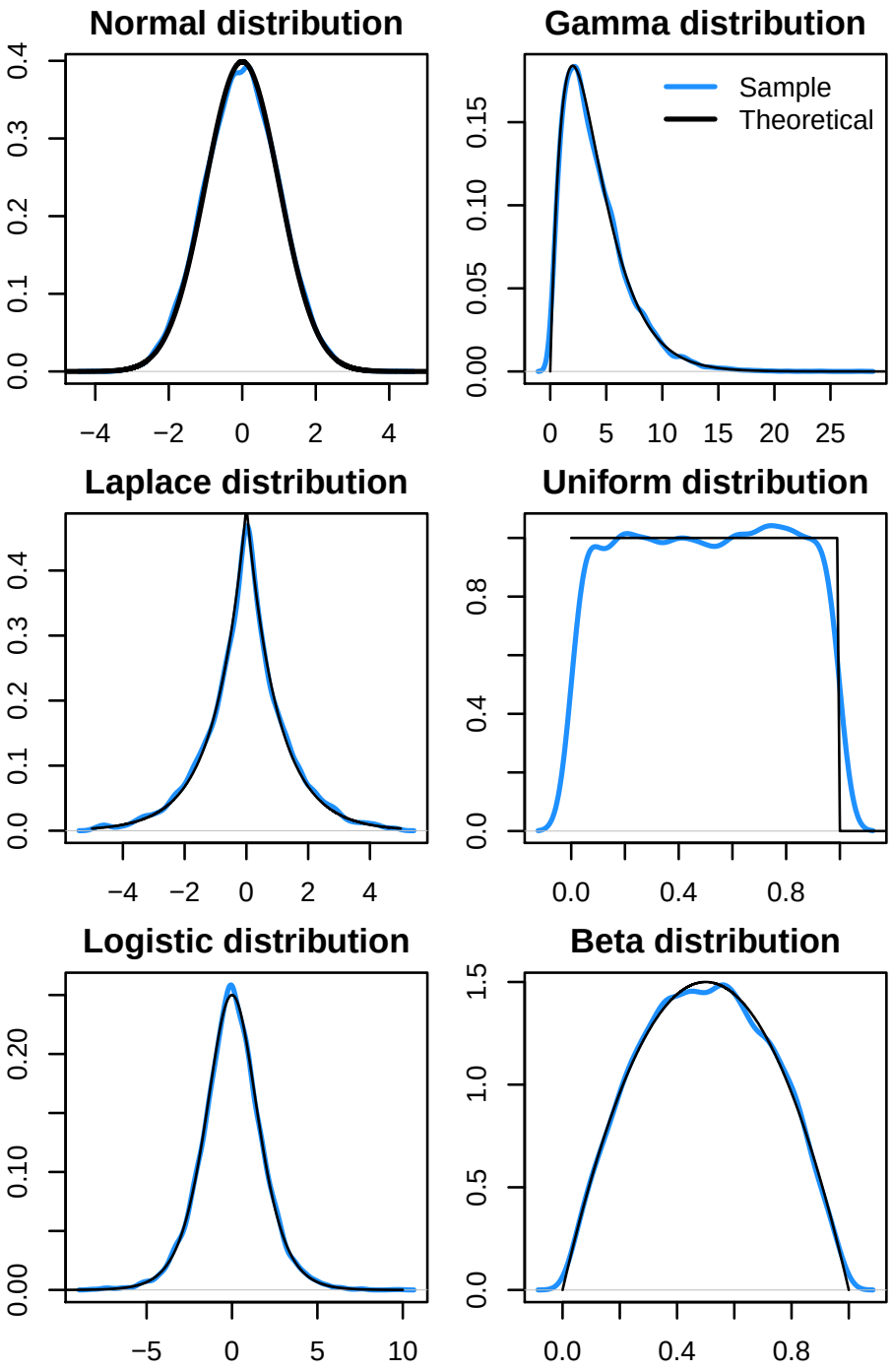

Figure 1: A comparison of the sample and theoretical pdfs for different log-concave distributions.