Adaptive Rejection Sampling

Jue Wang, Yang Li, Winnie Gao

12/08/2018

Email Address:

jue_wang@berkeley.edu, liyangc@berkeley.edu, zhaochen_gao@berkeley.edu

Github Repository: WinnieGao1223/stat243finalproject/

Please use this way to install our package:

devtools::install_github('WinnieGao1223/stat243finalproject/ars')

1. Approach

(a) Functions

Modular Functions:

- 1. Deriv_sub: Take the derivative of a function evaluated at x numerically. Basic logic: If x is at the boundary:

$$h'(x) \approx \frac{h(x) - h(x-\epsilon)}{\epsilon}$$

$$h'(x) \approx \frac{h(x+\epsilon) - h(x)}{\epsilon}$$

  Otherwise:

$$h'(x) \approx \frac{h(x+\epsilon) - h(x-\epsilon)}{2\epsilon}$$

- 2. check_concave: Check the log-concavity of the density functions. Check if the derivative of function h(x) decreases monotonically with increasing x.
- 3. init_mat: We create a matrix called mat to store the information that's collected during the rejection sampling process. The first column is the value of sample x. The second column is function h (which is log of original g function) evaluated at x. The third column is the derivative of function h evaluated at x. This function initializes mat, fills it with starting points provided by the user or starting points we find if they put -Inf or Inf.

```
init_mat <- function(a,b,fun,deriv){
  #this mat stores x,f(x) and df(x)
  mat <- matrix(nrow = 2,ncol = 3)
  mat[1,1] <- a
  mat[1,2] <- fun(a)
  if(!is.finite(mat[1,2])){
    stop('a is not defined on function h', call. = FALSE)
  }
  mat[1,3] <- deriv(a,fun,a,b)
  mat[2,1] <- b
  mat[2,2] <- fun(b)
  if(!is.finite(mat[2,2])){
```

```
    stop('b is not defined on function h', call. = FALSE)
  }
  mat[2,3] <- deriv(b,fun,a,b)
  return(mat)
}
```

- 4. update_mat: Update mat with x_star and let the first column in the ascending order, which is the proposed sample points generated by sampling process.

```
update_mat <- function(mat,x_star,fun,deriv,a,b){
  index <- sum(mat[,1] < x_star) #find the location to insert x*
  tmp <- matrix(NA,1,3)
  mat <- rbind(mat,tmp)
  mat[(index+2):nrow(mat),] <- mat[(index+1):(nrow(mat)-1),]
  mat[(index+1),] <- c(x_star,fun(x_star),deriv(x_star,fun,a,b))
  return(mat)
}
```

- 5. init_z: Initialize z, which is a function used to initialize a vector that contains different z values. The input will be the initial mat matrix which containes starting point a and b as well as h(a),h(b),h'(a) and h'(b). We let the first element in the z vector to be the starting point a, and the last element in the z vector to be the starting point b. We calculate the first z given a and b using the formula below.

$$z_j = \frac{h(x_{j+1})_h(x_j) - x_{j+1}h^{'}(x_{j+1}) + x_j h^{'}(x_j)}{h^{'}(x_j) - h^{'}(x_{j+1})}$$

Then we put this newly calculated z into the z vector and initialize the z vector.

```
init_z <- function(mat){
  z <- c()
  z[1] <- mat[1,1]
  z[3] <- mat[2,1]
  #if the slopes at a and b are equal, set z as the mean of a and b
  if (abs(mat[2,3]-mat[1,3]) < 1e-9){
    z[2] <- (mat[2,1] + mat[1,1])/2
  } else {
    z[2] <- calc_z(mat)
  }
  return(z)
}
```

- 6. update_z: update_z is a function we use to update the z vector with new z values.Each time we add a x value and recalculate the tangent line based on our current x value, we will get two new z points. we will add these two new z points in out vector. The location in vector z where we add these two new z points is determined by the magnitude of z relative to the existing z points.

```
update_z <- function(z,x_star,mat){
  if (nrow(mat) <= 2){
    stop("something wrong with info matrix", call. = FALSE)
  }
  index <- which(mat[,1] == x_star)
  z <- c(z,NA)
  tmp_mat1 <- mat[(index-1):index,]
  tmp_mat2 <- mat[index:(index+1),]
  if (sum(abs(diff(mat[(index-1):(index+1),3]))) < 1e-9){
    z1 <- mat[(index-1),1] + (mat[(index+1),1] - mat[(index-1),1])/3
```

```
    z2 <- mat[(index-1),1] + (mat[(index+1),1] - mat[(index-1),1])/3*2
  } else {
    z1 <- calc_z(tmp_mat1)
    z2 <- calc_z(tmp_mat2)
  }
  if((round(z1, digits = 6) > round(z2, digits = 6))){
    stop('something wrong with updated z value, check log-convexity!',
         call. = FALSE)
  }
  z[(index+2):length(z)] <- z[(index+1):(length(z)-1)]
  z[index] <- z1
  z[index+1] <- z2
  return(z)
}
```

- 7. sample_x: Generate a new proposed sample point. Here's the more detailed explanation with math formulas:

(1) To calculate the area under the specified segment of piecewise function, we use this formula:

$$p = \int_{z_{i-1}}^{z_i} e^{h(x_i)+(x-x_i)h'(x_i)}dx = \frac{1}{h'(x_i)} e^{h(x_i)+(x-x_i)h'(x_i)}\Big|_{z_{i-1}}^{z_i}$$

$$= \frac{1}{h'(x_i)}[e^{h(x_i)+(z_i-x_i)h'(x_i)} - e^{h(x_i)+(z_{i-1}-x_i)h'(x_i)}]$$

(2) We let p divided by sum of p to normalize p and get p_norm. We draw a random number from uniform distribution and compare it with cumulative sum of p_norm to determine which piece we want to sample from and use i to denote it.

(3) We draw another random number from uniform distribution to get the sample x through evaluating the inverse cdf at the value of random draw. In the following formulas, we use u to denote the random number. Here's the explanation:

$$\int_{z_i}^{x^*} \frac{e^{h(x_i)+(x-x_i)h'(x_i)}}{p}dx = u$$

$$\frac{1}{p_i h'(x_i)} e^{h(x_i)+(x-x_i)h'(x_i)}\Big|_{z_i}^{x^*} = u$$

$$\frac{1}{p_i h'(x_i)}[e^{h(x_i)+(x^*-x_i)h'(x_i)} - e^{h(x_i)+(z_i-x_i)h'(x_i)}] = u$$

$$e^{h(x_i)+(x^*-x_i)h'(x_i)} = up_i h'(x_i) + e^{h(x_i)+(z_i-x_i)h'(x_i)}$$

$$x^* = \frac{log(up_i h'(x_i) + e^{h(x_i)+(z_i-x_i)h'(x_i)}) - h(x_i)}{h'(x_i)} + x_i$$

```
sample_x <- function(z, mat, n=1){
  uk1 <- exp(mat[,2]+(z[-1]-mat[,1])*mat[,3])
  uk2 <- exp(mat[,2]+(z[-length(z)]-mat[,1])*mat[,3])
  p <- rep(NA,nrow(mat))
  p <- (uk1-uk2)/mat[,3]

  # normalize p
  p_norm <- p/sum(p)
  w <- runif(n)
  # determine the range we want to sample from
```

```
  i <- sum(cumsum(p_norm) < w) + 1

  # sample x using inverse cdf
  samp_x <- (log(p[i]*mat[i,3]*runif(n)+
                 exp(mat[i,2]+(z[i]-mat[i,1])*mat[i,3]))-mat[i,2])/mat[i,3]+mat[i,1]
  return(samp_x)
}
```

- 8. upper_bound: Calculate the upper bound. We take the z, mat and the newly generated sample x as input. We compare sample x with z to find which segment of line to calculate from, which is the index of smallest z that is larger than sample x. Then we use the linear relationship to get the upper bound.

$$u_k(x) = h(x_j) + (x - x_j)h'(x_j)$$

- 9. lower_bound: Calculate the lower bound. We take the mat and newly generated sample x as input. We compare sample x with first column of mat to find the index of largest x that is smaller than sample x and use the following formula to get the lower bound.

$$l_k(x) = \frac{(x_{j+1} - x)h(x_j) + (x - x_j)h(x_{j+1})}{x_{j+1} - x_j}$$

Main Part (ars): We try to find the appropriate starting point based on the information user provided and the log-concavity of the density function. We initialize mat and z vector. While the sample size does not reach our desired number, we repeat the rejection sampling process. First, we check the log-concavity of the density function every time we update the mat. Then we generate one proposed sample x using sample_x function. We draw one random number (w) from the uniform distribution and also calculate the upper bound (u) and lower bound (l) using corresponding functions. Then we determine if:

1. w <= exp(l-u), then accept the sample x and store it in the final result vector.
2. w <= exp(h(sample x)-u), then accept the sample x, store it in the final result vector, update the mat and z with this sample x.
3. Otherwise, reject the sample x and update the mat and z with this proposed sample x.

(b) Efficiency

- 1. Vectorization: We use matrix to store the updated information and we store z values and the x we sampled as vectors.
- 2. Updating Steps: Everytime we update the mat matrix and z vector, we make sure we keep them sorted in the ascending order so that we can easily find the position we want to insert the new proposed sample x.

(c) Special Case

For uniform distribution, since the derivative of h(x) is always zero, we use *runif* to draw random numbers and save them as the sample points.

2. Testing

(a) Input Checking

We wrote various tests to check if the inputs are valid, including check if the class of input function is actually a function, check if the starting points and sample size are numbers. We also checked whether the input function is log-concave and whther the function is defined on the given value.

(b) Unit Tests

We wrote the testing to check if two major modular functions (update_mat and sample_x) performs in the expected way. - 1. For update_mat: we test if the first column in the updated matrix is in an ascending order.

```
mat_test <- matrix(c(-2, 0, 2, h(-2),h(0),h(2),
                     Deriv_sub(-2,h,-2,2),Deriv_sub(0,h,-2,2),Deriv_sub(2,h,-2,2)), ncol = 3)
mat_update <- update_mat(mat_test,1,h,Deriv_sub,-2,2)
expect_equal(mat_update[,1], sort(c(1,mat_test[,1])))
if(sum(expect_equal(mat_update[,1], sort(c(1,mat_test[,1]))))){
  print("the x in the updated matrix is in an ascending orderl, update_mat unit test passed")
}
```

```
## [1] "the x in the updated matrix is in an ascending orderl, update_mat unit test passed"
```

- 2. For sample_x: we test if the sample is from the density we interested. We use standard normal distribution as our test case.

```
set.seed(5)
a <- 0
b <- 1
lambda <- 1
mat_test_2 <- matrix(c(a, b,log(dexp(a, rate=lambda)),
                       log(dexp(b, rate= lambda)),-lambda,-lambda), nrow=2)
z <- c(0, 0.5, Inf)
sample <- replicate(1000,sample_x(z,mat_test_2,1))
out.test <- ks.test(sample, pexp)
if(sum(expect_gt(out.test$p.value, 0.05))){
  print("sample matches exponential, sample_x unit test passed") } else{
  print("sample does not match exponential, sample_x unit test failed")
}
```

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

```
expect_gt(out.test$p.value, 0.05)
```

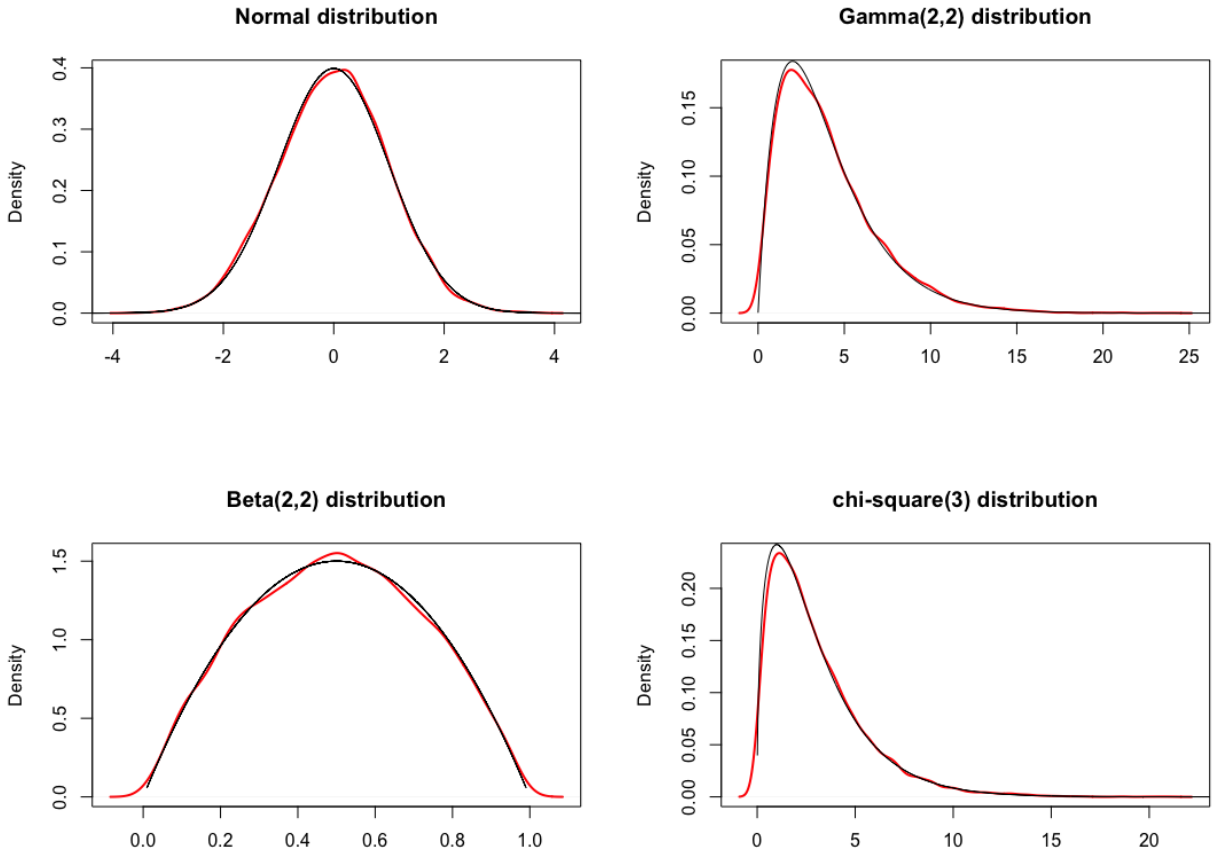
(c) Formal Testing

We checked several empirical distributions. We compared the samples we got from the adaptive rejection sampling algorithm we wrote with the samples we directly draw from the emprical distributions using the built-in functions in R. We use the ks.test() function in R to perform Kolmogorov-Smirnov test on the two samples we had in order to see if the two samples were drawn from the same continuous distribution. The log-concave empirical distributions we tested are: standard normal distribution, gamma(2,2) distribution, beta(2,2) distribution, and chi-square(3) distribution. In the Kolmogorov-Smirnov test, our null hypothesis is that the samples are from the same distribution. In all the test resuts we have in regards to the above

empirical distributions, the p-values are greater than 0.05. This means we fail to reject the null hypothesis. Therefore, we have done a pretty good job in terms of approximating the empirical distributions using adaptive rejection sampling. For the distribution that is not log-concave, we tested the e^{x^2} distribution. And our algorithm successfully gave us an error message.

(d) Graphical Comparison

The red curve is plotted using the samples we got from adaptive rejection sampling and the black curve is plotted using the samples we drew directly from the emipricial distributions.



3. Contributions

Yang, Jue and Winnie worked together to create the first draft of all the modular functions, and the main body of ars function. Then Jue started working on building the test cases for modular functions. And Yang contributed to the test cases included in the ars function. Winnie wrote the codes about how to find the appropriate starting points where f(x) is defined. Then Jue took care of coding about special cases like uniform distribution case and also the unit tests for important modular functions. Yang wrote the formal testing. Winnie wrote the overall help page and compiled the first draft of writeup. All of three members contributed to revisions. We worked together to make everything into a package.