

Stat 243 Final Project Writeup

Vaibhav Ramamoorthy, Colin Kou, Brandon Mannion, and Jonathan Lee

12/7/2018

Functions

We created a very modular solution for our Adaptive-Rejection Sampler, which handled unique subtasks within individual functions. Below, we list each function we created and its modular purpose:

get_initial_abscissae()

This function builds the vector of initial abscissae. If both endpoints of the inputted domain are finite, then the function simply uses those endpoints (plus/minus a small term to protect against closed intervals) as two of the abscissae and samples a third abscissae uniformly on the domain. If either of the endpoints is infinite, then the function iterates over values until it finds one whose first derivative is of the appropriate sign. For example, if D is unbounded on the right, we began at $x = 2$ and keep moving right until we find an x for which the derivative is negative. Again, once we have two finite abscissae based on the domain endpoints, we sample uniformly on the interval bounded by them to select the third abscissae.

This function is called at the beginning of `ars()` to set the initial abscissae.

get_z() and get_z_all()

Together, these functions use the abscissae and the log density to find the vector of points at which the tangent lines at the abscissae intersect. They utilize the equations presented in the Gilks, et. al. paper to calculate the vector of z 's. The `get_z_all()` function is called within `ars()` to recreate the z vector each time the abscissae get updated.

get_u_segment() and get_u()

Together, these functions use the abscissae and the log density to build the piecewise upper bound of the log density using tangent lines at x . Again, these functions utilize the formulas from Gilks, et. al.

We represent this piecewise function as a list of slopes and intercepts rather than as a closure-type variable in order to more easily calculate integrals later on. The `get_u()` function is called within `ars()` to recreate the upper bound each time the abscissae get updated.

get_l_segment() and get_l()

These functions work very similarly to `get_u_segment()` and `get_u()`, but instead build the piecewise lower bound by calculating the slopes and intercepts of the chords between adjacent abscissae, again using the formulas presented in Gilks, et. al.

Again, we represent this piecewise function as a list of slopes and intercepts. The `get_l()` function is called within `ars()` to recreate the lower bound each time the abscissae get updated.

get_s_integral()

This function calculates the integrals under each piecewise element of s , which is the normalized exponential of the u function. The function works by calculating the integral under each piecewise element of s analytically, using the formula as calculated below

$$\int_{z_1}^{z_2} \exp(u_j(t))dt = \int_{z_1}^{z_2} \exp(a_j + b_j t)dt = \int_{a_j + b_j z_1}^{a_j + b_j z_2} \exp(y) \frac{1}{b_j} dy = \frac{1}{b_j} [\exp(y)]_{a_j + b_j z_1}^{a_j + b_j z_2} = \frac{\exp(a_j + b_j z_2) - \exp(a_j + b_j z_1)}{b_j}$$

where a_j and b_j are the intercept and slope of the j^{th} segment of u respectively. The function returns a vector of integrals, of which the j^{th} element corresponds to the integral under the j^{th} segment of s .

sample.s()

This function performs the actual sampling. It samples n points from s by using the inverse transform sampling method as follows. We draw a vector q of length n from the $\text{Unif}(0, 1)$ distribution. Then, for each element of q , we find an x^* such that $F_s(x^*) = q$. To do so, we first have to identify the segment of u of which x^* is in the domain. Then, we calculate x^* by using the inverse CDF as follows:

$$\begin{aligned} \int_{z_j}^{x^*} \frac{1}{I_s} \exp(u_j(t))dt &= q - F_s(z_j) \\ \int_{z_j}^{x^*} \exp(a_j + b_j t)dt &= I_s(q - F_s(z_j)) \\ \int_{a_j + b_j z_j}^{a_j + b_j x^*} \exp(y) \frac{1}{b_j} dy &= I_s(q - F_s(z_j)) \\ [\exp(y)]_{a_j + b_j z_j}^{a_j + b_j x^*} &= b_j I_s(q - F_s(z_j)) \\ \exp(a_j + b_j x^*) - \exp(a_j + b_j z_j) &= b_j I_s(q - F_s(z_j)) \\ a_j + b_j x^* &= \log(b_j I_s(q - F_s(z_j)) + \exp(a_j + b_j z_j)) \\ x^* &= \frac{\log(b_j I_s(q - F_s(z_j)) + \exp(a_j + b_j z_j)) - a_j}{b_j} \end{aligned}$$

Using this formula as derived, we can convert our q vector into a vector of x^* . We return this vector of x^* s.

This function is called within `ars()` on different batch sizes to sample a vector of x^* s, which we then check for rejection or acceptance.

ars()

This is the main function of the package. It takes in a density function `FUN` and optionally, a sample size n , a domain D , and a verbose argument which if `TRUE` will print out information on how many samples were accepted and rejected in each batch iteration.

The `ars()` function makes use of all the other functions described above to initialize x and then calculate z , u , l , and sample from s during each batch iteration. After getting a sample using `sample.s()`, it performs the squeezing test to ascertain which x^* s to accept and which to reject/add to the abscissae.

[mention the check for log concavity]

One feature we added to `ars()` is that rather than sampling one value at a time from `sample.s()`, it samples a batch of size `batch.size` each iteration. The `batch.size` variable is initialized at 1 but is doubled each time all x^* s are accepted following the first squeezing test. This speeds up the sampling significantly.

Tests

Tests for Modular Components

We wrote tests for several of the modular components of our `ars()` implementation, as listed below:

- A test to check that the functions used to build the upper bound function outputted the correct slopes and tangent lines.
- A test to ensure that the function `get_s_integral()` outputted an integral that was larger than the integral of the density.

Kolmogorov-Smirnov Tests

To ensure the validity of samples generated from the `ars()` function, we ran a series of Kolmogorov-Smirnov tests which test for the closeness between an empirical CDF and a given distribution function in a nonparametric manner. We used the Kolmogorov-Smirnov test to test whether our `ars()` function returned a sample whose empirical CDF was close to the true CDF for a number of common log-concave distributions.

We ran the Kolmogorov-Smirnov test for the following distributions: Normal(0,1), Exponential(1), Beta(1,1), Beta(2,2), Gamma(1,1), Gamma(2,1), χ^2_2 , χ^2_3 , and Uniform(0,1), each at a significance level of 0.05. For each test, we made sure that the p-value was greater than the significance level, suggesting that the null hypothesis, which is that the sample is drawn from the reference distribution, was true. For all tests, the p-values were greater than the default significance level.

Team Member Contributions