Stat 243 Final Project Report

Brian Collica, Ashish Ramesh, Ben Searchinger

Location

The package, including all associated files and scripts, is located in the following public GitHub repository:

- bcollica/ars
- Link: https://github.com/bcollica/ars

The package can be installed locally using `remotes::install_github("bcollica/ars")`

Architecture

The code was written as a set of modular functions, each of which carries out a specific task. Each of these functions is then called by the main wrapper function `ars`.

Modular Functions

The functions we added include a function for generating tangent intersections for the given density from the abscissae provided, finding the upper hull from these tangent lines, sampling from the envelope created by the upper hull, and creating the lower hull from given abscissae. We included three additional helper functions: one for approximating a derivative via the finite difference method, one to comprehensively check the inputs to the main wrapper function, and one to periodically check that log-concavity is satisfied.

Below are the descriptions for each modular function:

- `tanIntersect()` calculates the tangent intersections to a given log-density, for tangent lines at a given set of abscissae. It does this by approximating the derivatives to the log-density at the given abscissae, and then using Equation (1) from Gilks et. al. to compute the x-coordinates of the intersections. This function returns a list containing the original abscissae, the x-coordinates of the interesections, and the values of the log-density and its derivatives at these intersections.
- `upperHull()` corresponds mainly to Equation (2) from Gilks et. al., and calculates the y-values for the upper hull of the log-density at a given set of x-coordinates. To do this, it first calculates the tangent intersections for the segments forming the upper hull, using the `tanIntersect()` function. Then, it uses the values returned by this function and Equation (2) to compute the y-value on the hull corresponding to any x-value provided as an input. This is done in a segment-wise manner, exploiting the fact that the intersections returned by `tanIntersect()` correspond to the boundaries of each segment in order to partition the x-coordinates given based on which segment they fall into.
- `sampleEnv()` corresponds most closely to Equation (3) from Gilks et. al., and samples `n` values from the upper hull for the log-density. It does this by first computing tangent intersections for a given log-density. Then, it finds the normalizing constant in Equation (3) by integrating numerically (using the `integrate()` function on `exp(upperHull(...))`). From here, samples are created using the inverse CDF method applied to each segment, using numerical integration to find the value of the CDF at each tangent intersection and explicitly computing the inverse of the CDF. The function includes a check to remove non-finite sample values and resample from the hull as needed.

- `lowerHull()` corresponds to Equation (4) of Gilks et. al., and computes y-values for the lower hull of the log-density. It does this by first calculating tangent intersections, and then uses Equation (4) to compute the appropriate y-values.
- `approxD()` approximates the derivative of a given input function at a specified point, using a central finite-difference method. It is capable of returning first or second-order derivatives, and uses a value of `sqrt(.Machine$double.eps)` as its default finite difference, although it allows a user to specify a different difference value.
- `checkThat()` serves as a comprehensive check of inputs to the main wrapper function. It includes checks for appropriate parameters, sample sizes, initial abscissae, and log-concavity of the density. It also generates starting values if none are provided.
- `cavitySearch()` checks for log-concavity of a given function at a single value or a vector of values. If the log-concavity condition is satisfied, the function returns `TRUE`.

The individual functions are designed such that they can be run independent of the main `ars()` function if needed. However, several of these functions include optional arguments that can accept the results of previous functions (for example, the outputs of `tanIntersect()`), which are used within the main `ars()` function to save computation time. Each function also has its own help page.

Main `ars()` Function

`ars()` takes five arguments, the details of which can be seen by calling `help(ars)` or `?ars`. These arguments give the user some flexibility. For example, the user can choose to supply some initial abscissae values, or have the function generate them instead. The arguments which are strictly necessary are: `n`, the number of values to sample; `f`, the distribution function; and (if applicable) `f_params`, a list of parameters to the distribution function. Additional optional arguments include a vector of starting abscissae and a 2-element vector for the support of the function (the default of this is $(-\infty, \infty)$).

The main function starts by checking the user arguments with `checkThat()`, generating starting values as needed. It then proceeds through the adaptive rejection sampling algorithm as described in Gilks et. al. In each computational step of the process, it calls the individual modular functions written for that task. The function returns a list containing the final sampled values, as well as counts of the number of abscissae added, the number of iterations, and the number of rejected samples.

Testing

We included tests for each individual modular function, as well as more comprehensive tests for the `ars()` function.

Testing for the helper functions `checkThat()` and `cavitySearch()` are mainly focused on catching known errors. The tests for `checkThat()` include checks to ensure that the function catches missing and/or incorrect arguments, and the test for `cavitySearch()` uses the t-distribution, which is known to not be log-concave, to ensure that it is caught as such. The tests for `approxD()` ensure that it correctly computes the derivative of a set of functions by comparing to known values, and checks for errors when incorrect parameters are used. The tests for functions related to computing the hulls (`tanIntersect()`, `upperHull()`, `lowerHull()`) check for correct dimensions of the outputs, as well as known mathematical properties. For example, there are tests for whether the upper and lower hulls bound the log-distribution correctly for two cases of inputs (normal and gamma distributions), as well as symmetry of the hulls and tangent intersections where appropriate.

Tests for `ars()` fall into 3 main categories. The first of these is checking reproducibility of results. A set of results for several distributions was generated using the same seed, and these tests ensure that, when run again with this seed, the `ars()` function produces the same outputs. The second set of tests involved testing for known errors. These tests ensure that non log-concave distributions and mismatched arguments are caught correctly. The final set of tests ensure that the function runs "silently" on inputs which are known to work.

Tradeoffs

One major tradeoff in our approach is due in part to the sequential nature of the algorithm. This meant that, while individual modular functions are vectorized, the best approach in following the algorithm itself ended up being to sample one value at a time. This is due to the computational cost of sampling values, meaning it's computationally expensive to over-sample and throw out the samples past the point at which it becomes necessary to recalculate `tanIntersect()`. Future versions of this package could revisit this tradeoff.

In testing the performance of our approach, we compared it to the existing `ars` package on CRAN. We noted that our results ran significantly slower. However, we believe this to be due to two main differences: the CRAN package uses C++ for the majority of its calculations, while our package uses R exclusively, and the CRAN package requires the user to pass in values for the derivative of the log-density, while our package includes the computation of these values and only requires the user to specify the distribution function and appropriate parameters.

Results

Our results for the number of iterations to achieve one sample from the standard normal distribution for different starting abscissae are shown below. The generating code can be found in `simulations.R`.

```
read.csv('./table.csv', sep = ',')
```

```
##      x1   x2 mean_iters max_iters mean_rej max_rej mean_abs_added max_abs_added
## 1 -10.0 10.0       4.47         8     3.47       7           4.05             7
## 2  -9.0  1.0       3.74         8     2.74       7           3.27             7
## 3  -8.0  2.0       4.03         6     3.03       5           3.49             5
## 4  -7.0  3.0       3.94         6     2.94       5           3.43             5
## 5  -6.0  4.0       4.00         7     3.00       6           3.40             6
## 6  -5.0  5.0       3.85         7     2.85       6           3.37             6
## 7  -2.0  2.0       2.70         5     1.70       4           2.26             4
## 8  -1.0  1.0       2.35         4     1.35       3           1.77             4
## 9  -0.5  0.5       2.77         7     1.77       6           2.36             6
```

Our results are generally consistent with Gilks et. al. with the caveat that we're counting iterations rather than evaluations of $h(x)$, so our results skew slightly lower. We also use 100 simulations instead of 1000 due to memory and time constraints, so there's more variability in our result. Our conclusion is the same, that the starting $(x_1, x_2) = (-1, 1)$ is ideal for the standard normal distribution.

Individual Contributions

Brian wrote the `approxD()`, `tanInteresect()`, `checkThat()`, and `cavitySearch()` functions, as well as their tests. He also worked on parts of the main `ars()` wrapper function. and its tests. Ashish wrote the `upperHull()` and `sampleEnv()` functions, as well as tests for these and the `lowerHull()` function. Ben wrote the `lowerHull()` function and the main wrapper `ars()` function, and the simulation that generated the data for the table. All group members were involved in catching errors in the various functions, planning the layout of the package, and writing the final report.