Stat 243–Intro to Computing
Final Project

Pi-Feng Chiu, Juanyan Li, Jamie Palumbo, Weiyan Shi

Due: December 17, 2015

1 Description of Code

1.1 Inputs

The sequential nature of the adaptive rejection sampling technique led our group to a very modular design approach. It follows that the main *ars* function is comprised primarily of calls to other functions in a specific sequential order. The *ars* function takes several inputs: a univariate, continuous, log-concave density function *f*, a lower bound *lb* and an upper bound *ub* for the distribution, the sample size *n*, a tolerance, and any additional arguments that might be necessary in specifying the distribution. If the function input is a user-defined function, the function they input must include *log* as a $TRUE$ or $FALSE$ argument. Before stepping into the heart of the ars calculation, the inputs are checked for validity. These validity checks stop the function from proceeding if it receives non-numeric inputs, non-positive sample size or tolerance, lower bound that is greater than the upper bound, or non-finite integration.

1.2 Initialization Step

Once the inputs pass these validity checks, the initialization step begins. First, the starting points, or abscissae, are chosen based on the mode of the given distribution and the bounds passed by the user. The *choose_init* function ensures that the starting points include the mode of the distribution in addition to one point to the left of the mode and one point to the right of the mode. These left and right hand side points are chosen to be relatively close to the mode and must be within the range of the upper and lower bounds of the specified distribution. While the user may input a value for the distance away from the mode each of these points will be, the default value, which worked very well in our testing, is 0.1. These starting points must pass the requirement for log-concavity before the function can continue. The *check_logconcave* function ensures that the second derivative at x is non-positive. Warning messages are returned if the point being checked is outside the lower and upper bounds, if the first derivative cannot be computed, or if the point is very close to the mode or the upper and lower bounds since these cases will return strange values. Next, *intersection*, *upperhull*, and *lowerhull* calculate exactly what their names imply based on the equations listed in the journal article and rudimentary algebra. The functions return the intersection points along with the slopes and intercepts of each of the line segments.

1.3 Sampling and Updating Steps

Having completed the initialization step, the sampling and updating steps may begin. Sampling will continue until a certain sample size is reached. The *sampleX* function finds a potential sample value that can be either accepted into the sample or rejected. The function itself integrates over each of the segment sections, dividing over the total area for normalization and calculating the cdf from these areas. A random sample from a standard uniform distribution is taken in order to select the segment to sample from and then solved for the sample value x using the inverse cdf method. The upperhull and lowerhull are then evaluated at the sample value x in *bndeval*. Finally, *srtest* performs the squeeze and rejection tests explained in the text and returns a boolean to indicate whether the sample value passed either of these tests. If the sample point is accepted, then it is added to the list of sample values. Otherwise, the value is added to T_k and a rejection test is run.

Each of the vectors is then updated according to whether the x value is inserted at the beginning, middle, or end of the vector.

2 Description of Testing

2.1 Unit Tests

Before testing the main *ars* function, we conducted several unit tests to ensure that our functions were working properly. These unit tests can be found in *test-unit.R* which is located in the tests directory.

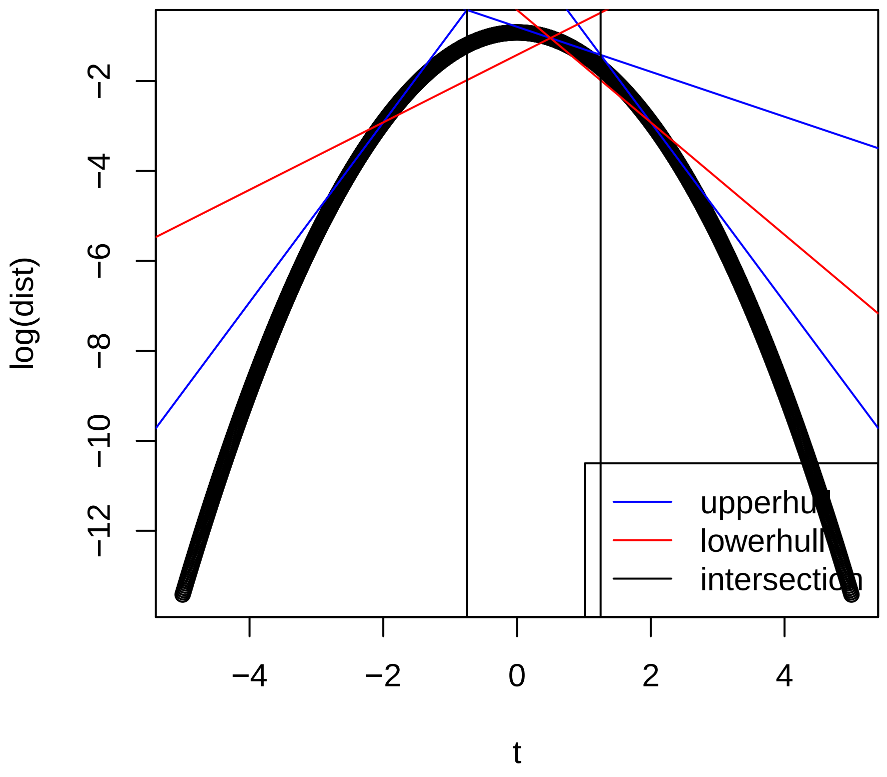
First, we tested the *choose_init* function, demonstrating three different cases that can occur and using dnorm as our model. The three cases are when the interval is unbounded, finite and covers the mode, and finite and does not cover the mode. We see that the initial points chosen for each of these cases are within the boundary limits and are, thus, valid starting values. Therefore, this function appears to be performing as expected.

Second, we tested the *check_logconcave* function, again using dnorm as our model for the majority of the tests in addition to an exponential function. The *check_logconcave* function found that log-concavity held true for dnorm at its boundaries when these boundaries were defined to be infinite, finite and covering the mode, and finite and not covering the mode. Log-concavity also held true at values close to the mode for each of these cases. We also tested a case in which the function was not log-concave. Our function returned a boolean to indicate that e^{x^2} was not log-concave at a certain point. For the purposes of testing, we withheld any error or warning messages that our function produced since they were coded into our function and were working properly. Overall, this function appears to be performing correctly.

Third, we performed a unit test for the *update* function. We considered placing the new sample point at the beginning, middle, and end of the vector. We found that in each of these cases, the new element was placed in the correct location. Therefore, this function also appears to be performing correctly.

Fourth, we performed a unit test to encompass the *upperhull*, *lowerhull*, and *intersection* funtions. Performing each of these functions on a test case, we graphed the outputs and found that the line segments for both the upperhull and lowerhull in addition to the intersection points appear to be calculated correctly. Therefore, we are confident that these functions are performing as expected.

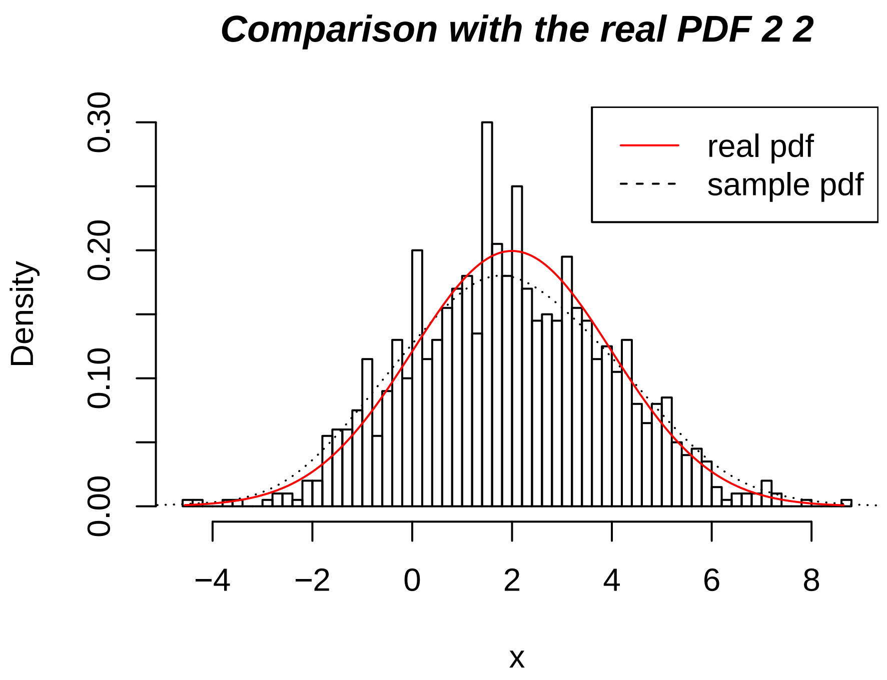
```
## [1] "########################################################################"
## [1] "This is a unit test for the upperhull/lowerhull/intersection functions."
## [1] "########################################################################"
## [1] "Please see the plot."
```



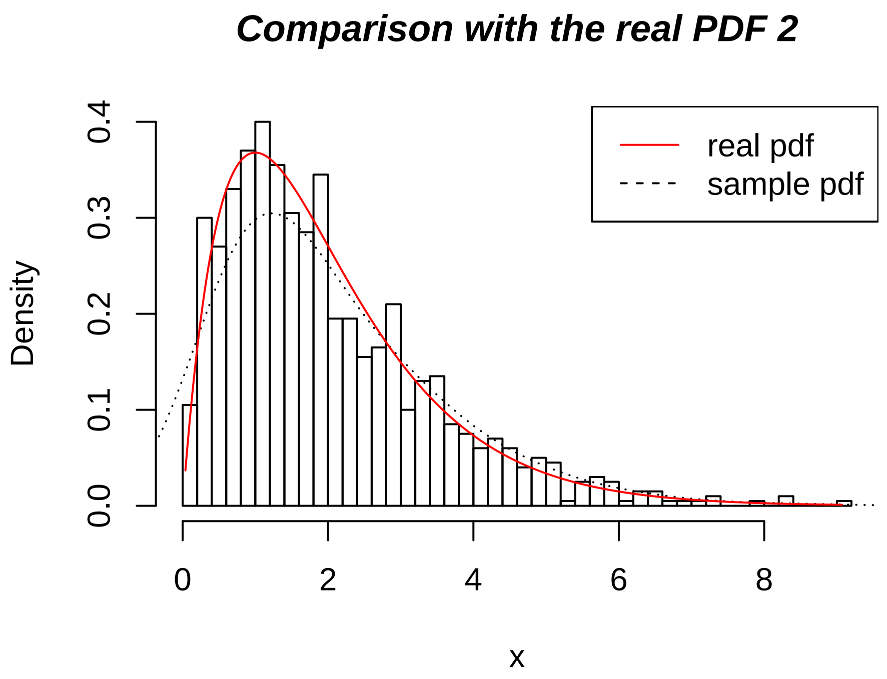
2.2 Main Tests

Once we felt confident that our helper functions were running properly, we began our main test of the *ars* function. This main test can be found in *test-main.R* which is located in the tests directory. We compared our sampled distribution to the true distribution for the functions dnorm, dgamma, dbeta, dlogis, dexp, dchisq, dunif, and dweibull. For each distribution, we input bounds that included the entire domain and bounds that did not cover the mode to ensure that both cases ran smoothly. Each test generated a plot with our sampled pdf, true pdf, and the histogram of our sample. In each of the cases, our sampled pdf seemed to very closely follow the true pdf. We have included examples for each distribution including the entire domain below to illustrate how well our main tests run. For more examples, please refer to the tests file.

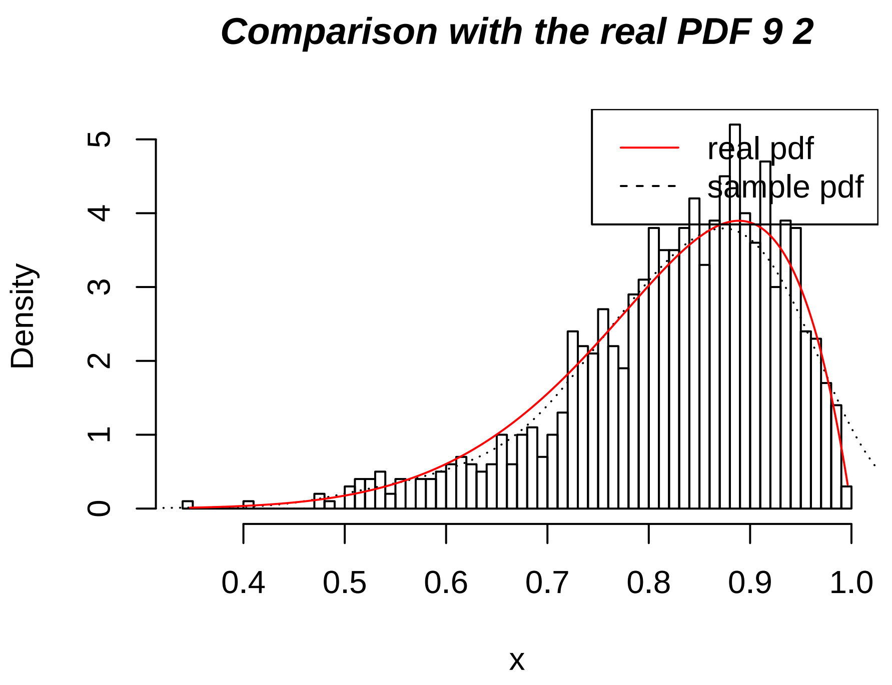
```
## [1] "######################################################################"
## [1] "This is the main test for the ars package."
## [1] "######################################################################"
## [1] "Demonstrate five cases of ars function."
## [1] "=======**********__________dnorm_____________*********============="
## [1] "Case 1: dnorm; mean=2,sd=2,lowerbound=-Inf,upperbound=Inf(the entire domain)"
## [1] "The inital values are:"
## [1] 1.9 2.0 2.1
```



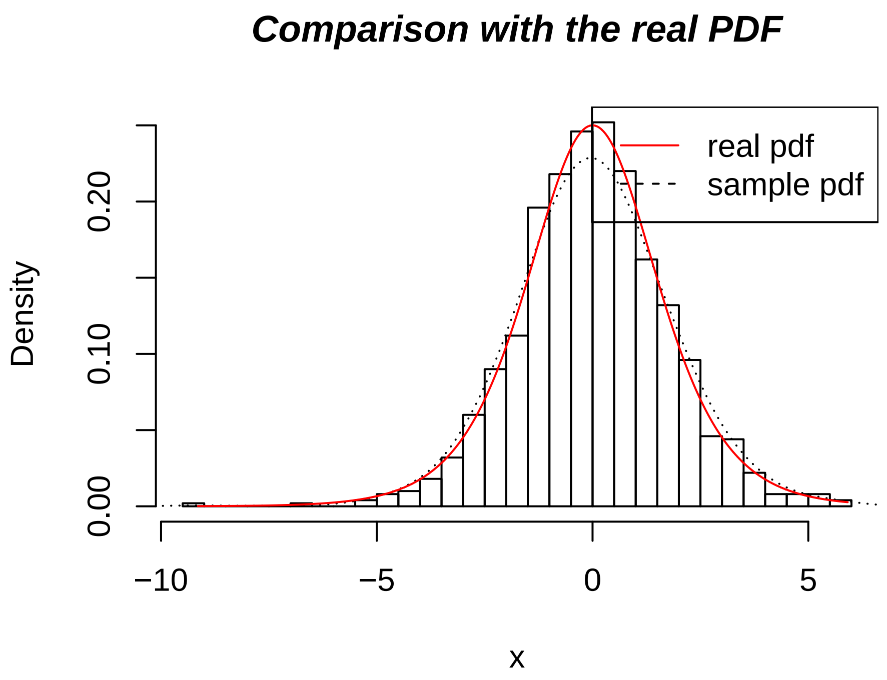
```
## [1] "===================**********__________dgamma_____________*********=============="
## [1] "Case 1: dgamma; shape=2,lowerbound=0,upperbound=Inf(the entire domain)"
## [1] "The inital values are:"
## [1] 0.9000023 1.0000023 1.1000023
```



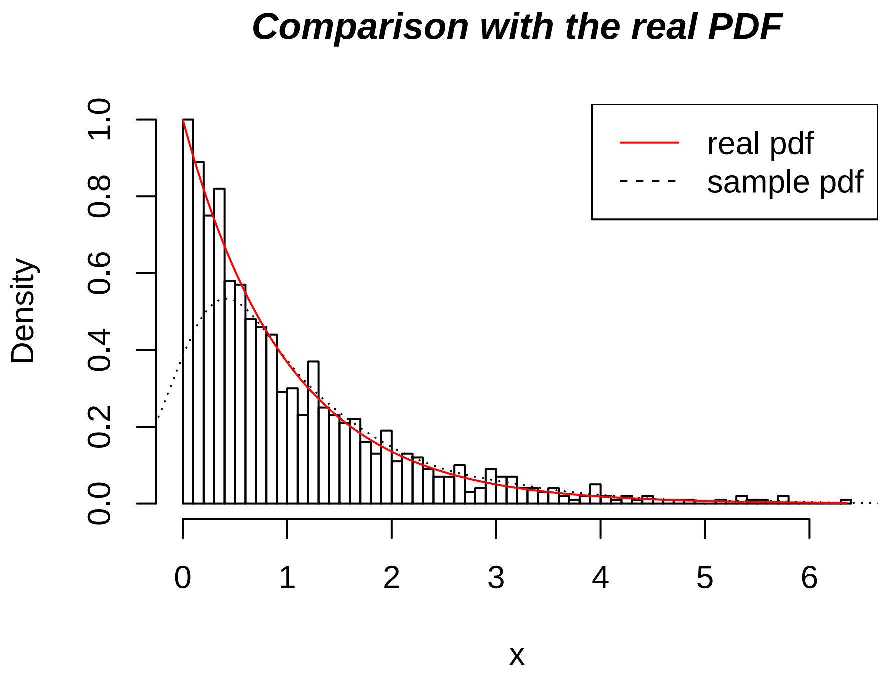
```
## [1] "===================**********__________dbeta_____________*********************="
## [1] "Case 1: dbeta; shape1=2,shape2=2, lowerbound=0,upperbound=1(the entire domain)"
## [1] "The inital values are:"
## [1] 0.7888989 0.8888989 0.9888989
```



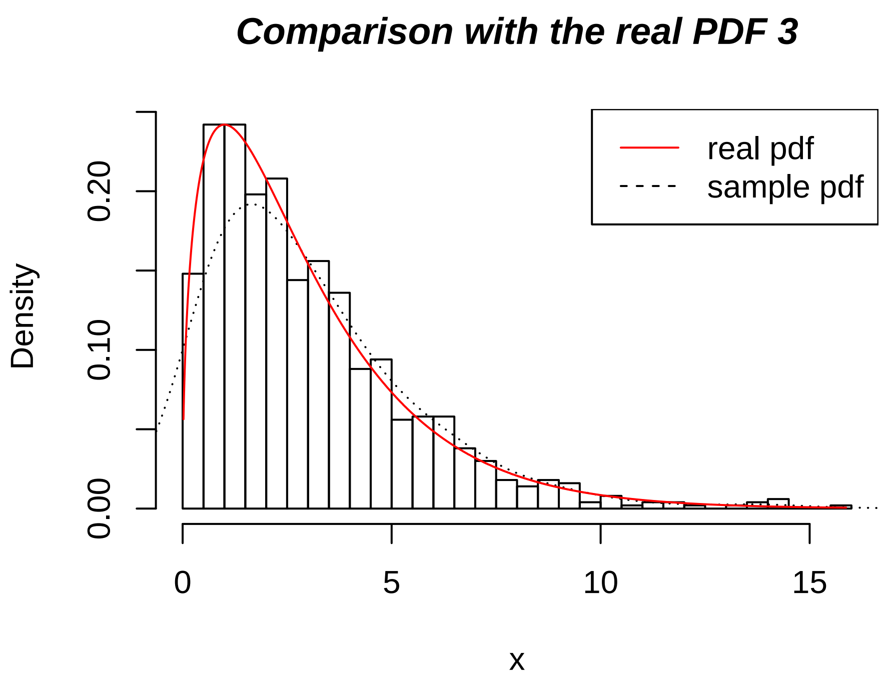
```
## [1] "===================**********__________dlogis_____________*********************="
## [1] "Case 1: dlogis; scale=1,lowerbound=-Inf,upperbound=Inf(the entire domain)"
## [1] "The inital values are:"
## [1] -0.1000142385 -0.0000142385  0.0999857615
```



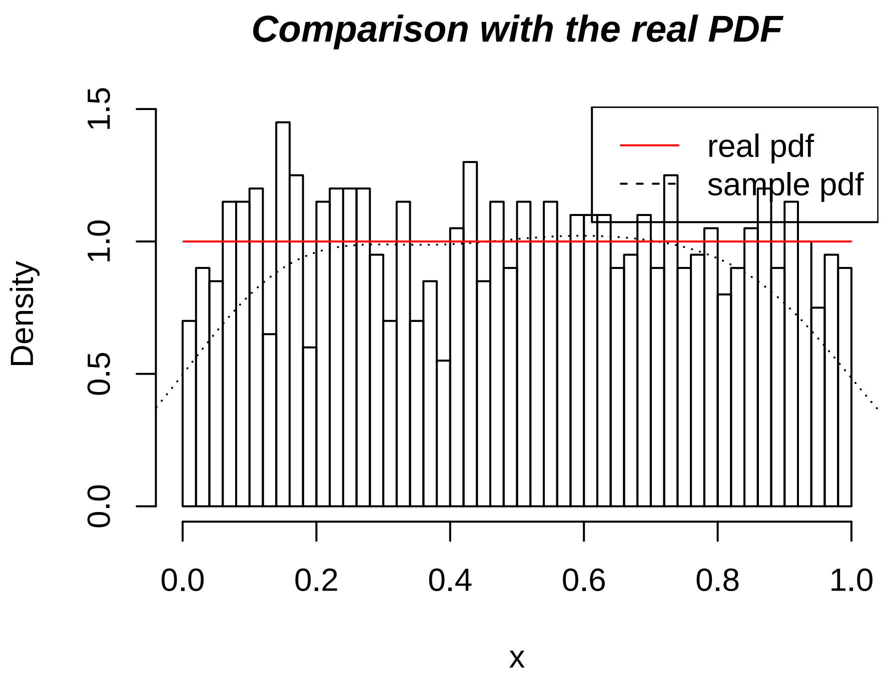
```
## [1] "===================**********__________dexp____________*********=============="
## [1] "Case 1: dexp; rate=1,lowerbound=0,upperbound=Inf(the entire domain)"
## [1] "The inital values are:"
## [1] 0.0000485809 0.0001220703 0.1000485809
```



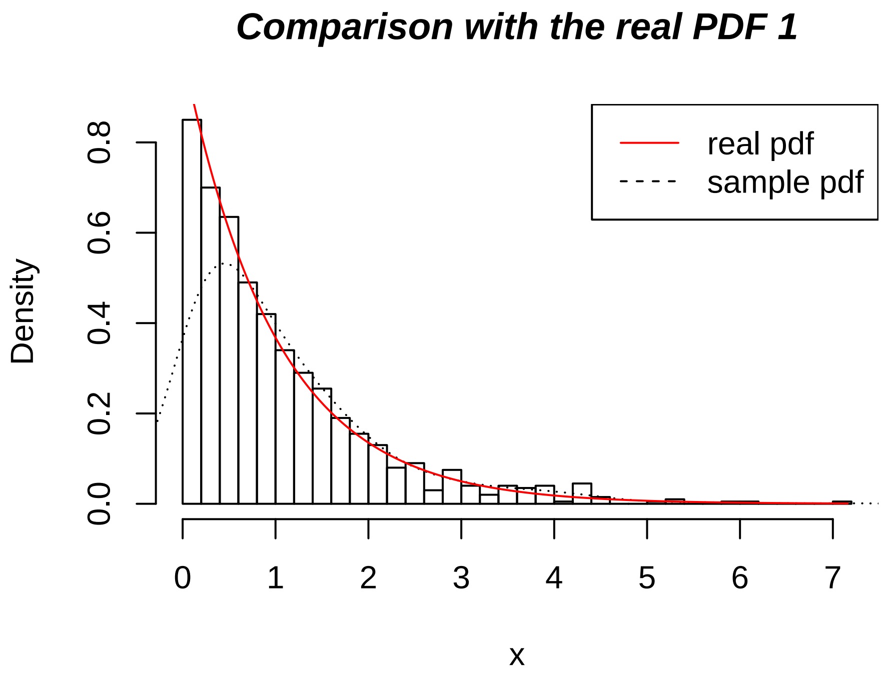
```
## [1] "===================**********__________dchisq_____________*********************="
## [1] "Case 1: dchisq(df=3); lowerbound=-Inf,upperbound=Inf(the entire domain)"
## [1] "The inital values are:"
## [1] 0.9000004 1.0000004 1.1000004
```



```
## [1] "===================**********__________dunif_____________*********************="
## [1] "Case 1: dunif(0,1),lowerbound=0,upperbound=1(the entire domain)"
## [1] "The inital values are:"
## [1] 0.8999339 0.9998779 0.9999339
```



```
## [1] "===================**********__________dweibull_____________**********************="
## [1] "Case 1: dweibull,shape=1,lowerbound=0,upperbound=Inf(the entire domain)"
## [1] "The inital values are:"
## [1] 0.0000485809 0.0001220703 0.1000485809
```



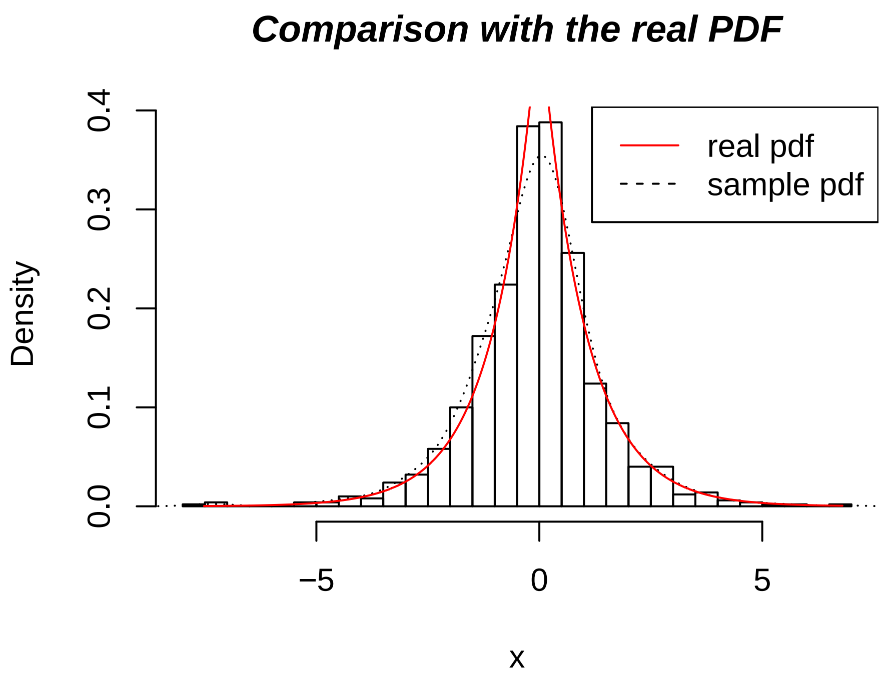
```
## [1] "PASS!"
```

Additionally, we tested dbeta in which the upperbound was not legitimate, and we tested dpareto in which the density function was not log-concave. Both of these cases returned an error message, catching the error and ending evaluation of the function as shown below. Additional non-log-concave functions may be found in the test file itself.

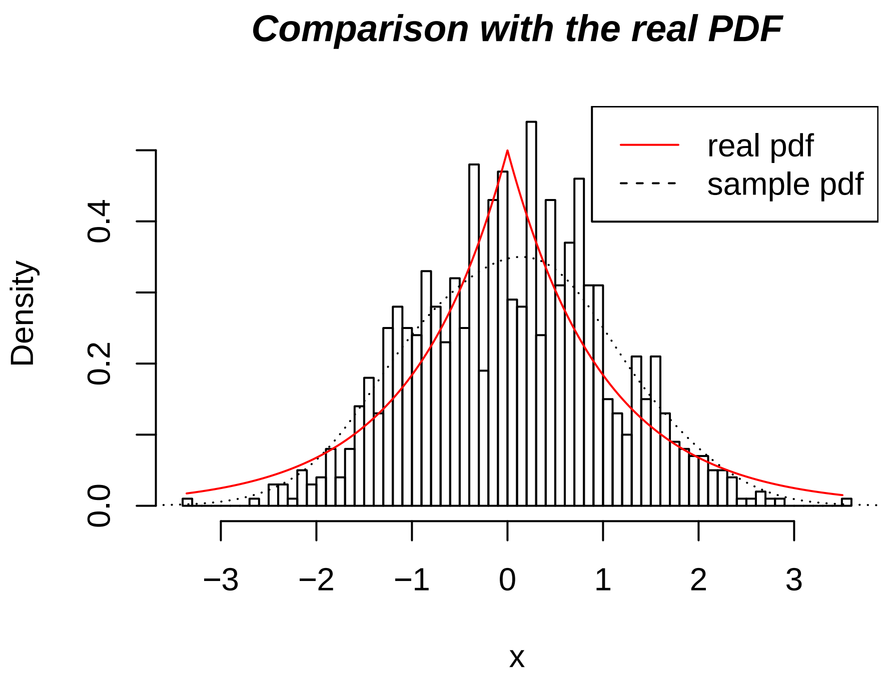
```
## [1] "----------------------------------------------------------------------"
## [1] "Case 1: dbeta; shape1=9,shape2=2, lowerbound=0,upperbound=Inf(the upperbound is not legitimate,

## Error in ars(dbeta, n = 100, shape1 = 9, shape2 = 2, lb = 0):  not able to compute the mode!
Please give the legitimate upper and
##          lower boundary for the density

## [1] "===================*********__________dpareto_____________*********************="
## [1] "Case 2: dpareto is not log-concave, and we catch the error!"

## Warning in check_logconcave(f, left, maximum, lb, ub, ...):  Not able to check logconcavity
at x because x is too close to the maximum of f

## [1] 1.100049

## Error in check_logconcave(f, right, maximum, lb, ub, ...):  The density function is not
log-concave at the point shown above!
```

Finally, we have also included tests of user-defined functions, including the Laplace and Subbotin distributions. Again, we will reiterate that user-defined functions must have an argument indicating whether log is $TRUE$ or $FALSE$. The results of these tests are found below.

```
## [1] "===================*********__________dpareto_____________*********************="
## [1] "Case 1: dlaplace; lowerbound=-Inf,upperbound=Inf(the entire domain)"
## [1] "The inital values are:"
## [1] -9.999495e-02  5.054477e-06  1.000051e-01
```

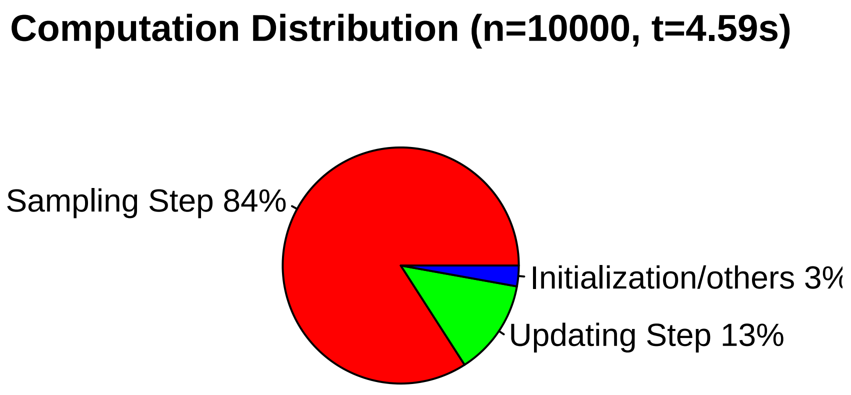


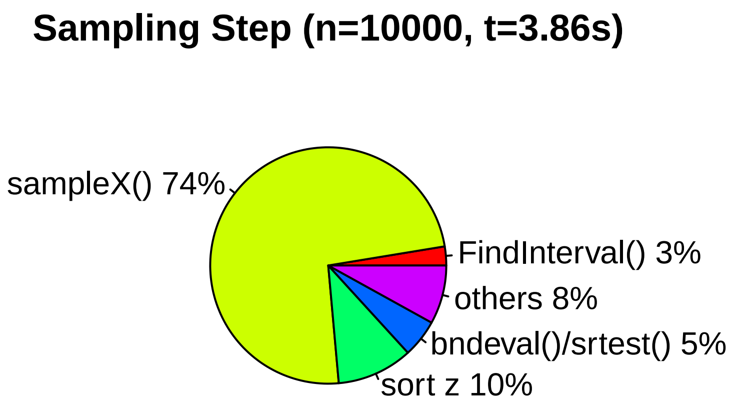
```
## [1] "===================***********__________dsubbotin_____________*********************"
## [1] "Case 1: dsubbotin; lowerbound=-Inf,upperbound=Inf(the entire domain)"
## [1] "The inital values are:"
## [1] -1.000000e-01  1.164153e-10  1.000000e-01
```



2.3 Timing

Lastly, we checked the computational efficiency of our code. For a sample size of 10,000 and the dnorm distribution, we found that our code took approximately 4.59 seconds to run. The sampling step appeared to take up the majority of this time at 84%, followed by the updating step at 13%, and then the initialization step at 3%. The function that mainly affected the timing was the *sampleX* function coming in at 2.85 seconds which was approximately 74% of the sampling step. This is probably due to the fact that we must call this function every single time to get a new sample value, and this sample value may not even be accepted into our overall sample.





3 Team Contributions

Each member of the team contributed fairly to the project. Each member was involved in developing the stucture of the code, debugging errors, improving efficiency, and carrying out tests. Since the code was modular in nature, the steps involved in adaptive rejection sampling were divided among the team members. In particular, Pi-Feng created the functions involved in calculation of the envelope and squeeze in addition to the updating step. Juanyan created the functions involved in acceptance and rejection of the sample value. Jamie created the sampling function and wrote up the pdf of the solution and the help page for the ars function. Finally, Weiyan created the functions to determine the initial starting values and check for log concavity.

4 Github Respository

The final version of the project can be found in Pi-Feng Chiu's respository at pfchiu.