SDS 394 Project Update

Carlos Gillett (ctg576) & Kayla Schaefer (khs426)

Spring 2015

1 Topic Recap

We are making our project on Approximate Bayesian Computation, specifically looking at what has been deemed "Tiny Data". We are focusing on a concrete example for our project: estimating the number of socks a person has given a random sample of size < 15.

The ABC algorithm works as follows [1]:

1. Construct a generative model that produces the same type of data as you are trying to model. Assume prior probability distributions over all the parameters that you want to estimate.
2. Sample tentative parameters values from the prior distributions, plug these into the generative model and simulate a dataset.
3. Check if the simulated dataset matches the actual data you are trying to model. If yes, add the tentative parameter values to a list of retained probable parameter values, if no, throw them away.
4. Repeat step 2 and 3 to build up the list of probable parameter values.
5. Finally, the distribution of the probable parameter values represents the posterior information regarding the parameters.

2 Project Goal

Our goal was to let the user can input how many individual and paired socks were in the sample, what prior parameters you want to use, and how many threads over which to parallelize. Then we run the ABC algorithm and return estimates for how many socks you have as singles, as pairs, and overall, along with the proportion of single socks. We also create histograms for the probability distributions of those estimates.

3 Program Functions

3.1 I/O

The I/O for the program allows the user to either run a demonstrative example or set as many of the inputs as desired. An -h flag is also implemented to output a helpful description of all accepted inputs.

```
> ./abc.exe -h
options:
-u: count of unique socks
-p: count of paired socks (e.g. 1 pair = 2 socks)
-m: estimated amount of total socks
-s: error for total socks estimate (default is m / 2)
-n: 'small' or 'large' for proportion of paired socks
```

3.2 Prior Parameters Construction

For the prior parameters, an estimated mean and standard deviation are accepted from the command line, and used to calculate the p and r parameters necessary for the negative binomial distribution over the total number of socks. These were derived from

$$\mu = \frac{r(1-p)}{p}$$

$$\sigma^2 = \frac{r(1-p)}{p^2}$$

Simple rearrangement and substitution gives us

$$p = \frac{\mu}{\sigma^2}$$

$$r = \frac{\mu p}{1-p}$$

For the prior on the proportion of paired socks, the user simply inputs either "small" or "large", to set the starting median (either .9 or .95), as we believed that was more intuitive for the average user.

After entering inputs (or not), the program will print out all the parameters used in the current run of the executable.

```
Using:
11 unique socks
0 paired socks
30 estimated total socks +- 15 socks
small proportion of paired socks
```

3.3 Simulation Function

The simulation function, sock_sim.c or sock_parallel.c, returns the desired number of samples from the prior distributions. It takes in the prior parameters, observed information, number of samples, and the pointer to a counter variable to keep track of the number of samples that match the observed data. The function is a large loop over the number of iterations specified in the original function. For each time through the loop, the samples are generated using the GSL library's random distribution functions as follows, where r represents a seed:

- nsocks = gsl_ran_negative_binomial(r,p,n);
- pairProp = gsl_ran_beta(r,alpha,beta);
- npairs = floor(nsocks*pairProp);
- nodd = nsocks - 2*npairs;

Once we have the sample population is specified, we construct this population. The population is represented as a vector of ascending integers, with matching socks having the same integer. For instance, a sample of 11 socks with 3 single socks would appear:

$$[0, 0, 1, 1, 2, 2, 3, 3, 4, 5, 6]$$

This a sample of this population was then generated, which matches the size of the observed sample input into the program. The sample is drawn using the gsl_ran_choose function, which takes a generates a random subset of a vector. Then, using a simple loop over the vector, the number of paired and single socks in the sample are counted. Finally these results are compared to the observed results, and the match_count variable and vector of results are adjusted accordingly.

The results vector stores, for each iteration of the loop, the sampled number of socks, pair proportions, pairs and odd socks, as well as a binary indicator of whether the generated sample matches the observed sample. After the loop is completed, the simulation function returns the number of matches to the main function. The results vector is passed in so that the changes made to it in the simulation function are represented in the main function, and no information needs to be formally passed.

3.4 Post-simulation Processing

After the number of matched samples is returned to the main program, vectors are allocated to collect the sample results. These are collected from the large vector function in a loop. The loop takes steps of 5, noting whether or not the matching sample indicator is 0 or 1. If it is one, the appropriate values are copied into the corresponding vectors. These vectors are then written to individual .dat files, using the printVector function. The printVector function intakes the vector, file name, and size, and writes the results to the specified file.

The last functions taking place within the program are the calculation of the estimates and the printing of these findings to the screen. We use the median of our posterior data for our estimates. The median for each final result vector is done in two steps. First, the vectors are sorted using the gsl_sort function. Then, we calculate the median of each vector using the gsl_stats_median_from_sorted_data function. These results are then printed to the screen, as shown below.

```
Estimated number of socks: 39
Estimated number of pairs: 17
Estimated number of unmatched socks: 5
Estimated proportion of paired socks: 0.880
```

3.5 Plotting Results

The hist_plots.py code takes the .dat files output by the main function and generates a .png file showing the posterior distributions of each variable. Python's matplotlib library was used for the hist() function. Each plot is a subfigure and then combined into a single .png. An example figure is shown below.

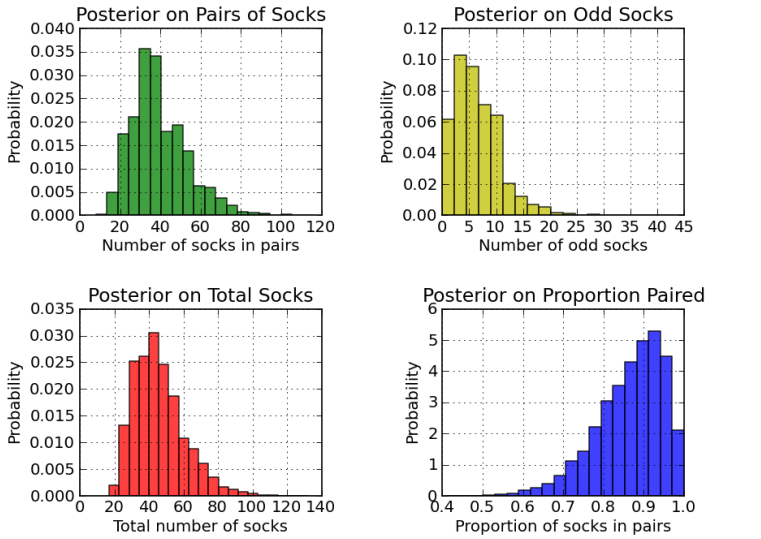

Figure 1: These are the posterior plots for the example case

4 Profiling

We have profiled the code using Tau. However, the code doesn't take too long to run. For instance, with one million iterations the time command counts 1.553 seconds of real time. Considering the practical application of our data, there is not a huge advantage to running more than this number of iterations, so for most intents and purposes it is fast enough as is. However, we have profiled our code using tau and paraprof to practice this technique. When profiled, the code used ten thousand iterations and the default inputs. With these settings the function in total spends just over 0.11 seconds in the main function, 0.08 seconds in the simulation function, and 0.031 in the print-vector function. The figures of these results can be seen in Figure 2 below.

The simulation function uses roughly 4.53 million floating point operations, with the main function having 200. The printing function uses just 24. The plots of these results can be seen in Figure 3 below.

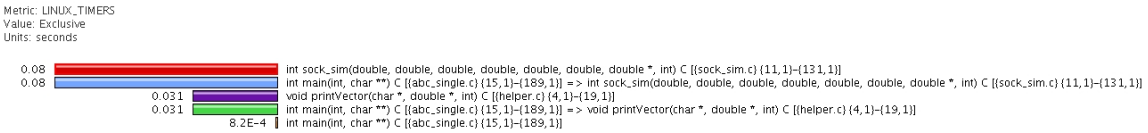

Figure 2: Time in each function

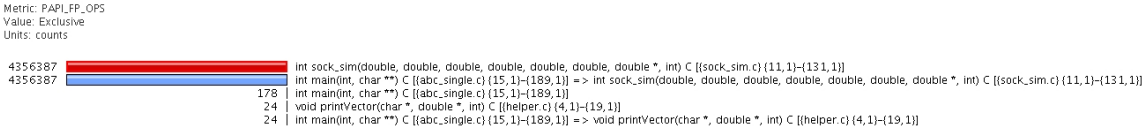

Figure 3: Floating point operations in each function

5 Parallelization

The code was put into parallel using Open MP. Originally, we had intended to use MPI, but Open MP was selected because we wanted to parallelize components of the program, but not run all processes separately. Furthermore, MPI does not easily allow for paralyzing sections of a program, which is something Open MP excels at. Because each element of the array is accessed separately, sharing memory is not an issue. Thus Open MP is initialized in the simulation function, splitting the large loop over different processors. At the same time, given the short runtime of our serial version, the user will not see a noticeable speedup with the parallelized executable.

6 Conclusion

Our goals for this project were to write C code to allow the user to provide inputs for a simulation with the ABC algorithm. Then we run ABC and return estimates for how many socks you have as singles, as pairs, and overall, along with the proportion of single socks, taking into account the user's data. We save the resulting samples to file and create histograms for the probability distributions of those estimates. This was all run on lonestar, compiled with icc, and uses a makefile. The code has all been uploaded to bitbucket using git. The instructions for building and running these codes are in the README. We fulfilled additional requirements by also including a parallelized option and by analyzing the performance of our code with the Tau profiler and using the GSL scientific library.

References

[1] Baath, R. (2014). Tiny Data, Approximate Bayesian Computation and the Socks of Karl Broman. Retrieved from http://www.sumsar.net/blog/2014/10/tiny-data-and-the-socks-of-karl-broman/.