

Progress report for an efficient implementation of heuristic partition function calculations in C

Rachel-Anne Arthur, Ashton Berger, James Corbin II, & Willie-Man

The University of Texas at Austin

Austin, TX 78712, USA

November 20th, 2015

I. Introduction and concept

Our project is centered on developing an efficient algorithm for the calculation of a modified partition function for various ligand-receptor pairings in a multi-species ligand and multi-species receptor system. We assume that the systems will range from $L=1$ to $L=10000$ unique ligand species and $R=1$ to $R=10000$ unique receptor species. Given a maximum possibility of $L \times R = 10000 \times 10000 = 100000000$ ligand receptor pairings, and therefore an equal number of maximum partition function calculations to be performed, this endeavor has potential to be very computationally expensive. This report is an update on how we have decided to approach the problem, what progress has been made towards a solution, and what we plan to do in the near future in order to realize our projects success.

We have data files that serve as the input for the program. The data files are tables stored in a .csv format, and an example of the data can be seen below on the following page. We will keep the formatting of the output to be the same as the input to maintain consistency. We decided our software would need the bare minimum capabilities in order to achieve our goal: input file handling, data storage of the input values from the input file, functions to perform the calculations on the stored input values and then replace them with the new calculated values, and output file writing. We plan on utilizing parallel programming techniques within the functions that perform the calculations, and we also plan on using Make to build the executable files for running. Git version control is allowing us all to collaborate on the software together while keeping track of bugs, errors, and documentation. Once we have the program running properly for small data files (e.g. L & R less than 20), we will begin running it on much larger cases and determine where else to implement other scientific programming techniques and optimizations.

II. Progress

i. File input

The input function is written to import a .csv file and parse it into several pieces of data including the main data structure which is an $m \times n$ matrix. The input function is written

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	400																				
2		L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13	L14	L15	L16	L17	L18	L19	L20
3	R1	-400	0	0	0	-30	0	0	0	-145	0	0	0	-20	0	0	0	0	-15	0	0
4	R2	0	-500	-310	0	0	0	0	0	0	-50	0	0	0	0	0	-15	0	0	0	0
5	R3	-231	0	-650	0	0	0	0	-308	-162	0	0	0	0	-58	0	0	0	0	0	0
6	R4	0	0	0	-200	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	R5	0	0	0	0	-150	0	0	0	0	0	-68	-168	0	-20	0	0	-52	0	-68	0
8	R6	0	-112	0	0	0	-405	0	-324	0	0	0	0	-26	0	0	-14	0	0	-81	0
9	R7	0	0	-89	0	0	-398	-300	0	0	0	0	0	0	0	0	0	0	0	0	0
10	R8	0	0	0	0	0	0	0	-200	0	0	-91	0	0	0	0	0	0	0	0	0
11	R9	0	0	0	0	0	0	0	0	-510	0	0	0	0	-83	0	0	-44	0	0	0
12	R10	-41	0	0	0	0	-76	0	0	0	-400	-92	0	0	0	0	-69	0	0	0	0
13	R11	0	0	-148	0	0	0	0	-316	0	0	-300	0	0	0	0	0	0	0	0	0
14	R12	0	0	0	0	0	0	0	0	-260	0	0	-530	0	0	0	0	0	0	0	-351
15	R13	0	0	-51	0	0	0	0	0	0	0	0	0	-440	0	-8	0	0	0	0	0
16	R14	0	0	0	0	0	0	0	0	0	0	0	0	0	-430	0	0	0	0	0	0
17	R15	0	0	0	-271	0	0	0	0	0	-348	-192	0	0	0	-540	0	0	-271	0	0

Figure 1: An example table with $L=20$ and $R=15$. The actual data will not be integers, but instead floats with 6 decimal-place precision.

into the code of the main function with one external supporting function that converts ascii to a double precision number accurate to 6 decimal places.

The input function begins by asking the user to input the name of the csv file to be processed and stores in a string array. It also asks the user to input the number of ligand columns and receptor rows in the data file. It stores this information in global variables to be used by all functions. The input function then opens the data file and copies into an $m \times n$ element array and then closes the file. It is important to note that the data file to be processed must reside in the same directory as the executable. Then we begin parsing the array containing the data.

The input data file must conform to a couple other typical formatting standards. The very first piece of data needs to be the temperature to be included in the processing calculations. Also, there needs to be column and row headers that will be discarded. The ascii representation of the temp data is copied into a working array. The array is then passed to the ascii to float conversion function. The result is returned and stored into the global temperature variable (T). Assuming the very next line will be all column headers, it will be discarded. Next we will parse all the data. The row header will be discarded and each data field will be passed to the ascii to float conversion function. The returned values will be written in exactly the same column/row location that they occupied in the input data file. The matrix is a global variable and therefore can be used by all functions for reading or writing.

The main function containing the input file function and its helper function which converts ascii to float are essentially complete. The main function still needs to have the overall algorithm completed as it can only currently execute the input function. As soon as everyone completes their portions and functions, calls to them and the final algorithm will be written and tested. We estimate that this remaining code will be short and concise as the bulk of the code will be contained in functions. Even though the input function is fully operational, James would like to improve some of its functionality. For example, we decided that we would limit the functionality of our program to matrices of no larger than 10000 by 10000 elements. James initialized an input file and data array capable of handling the maximum. James intends to change this to a memory allocation (malloc) so that we can make the

sizes of both of these data structures variable. Most likely this will increase the efficiency of the memory usage as we don't expect the maximum value to be used often. James is also considering writing a loop that will check whether an input file has column headers rather than requiring them to be present.

ii. Calculations

The values in the input table are all the variables that are required during the calculations. As described earlier, the very first value in the top left corner is the temperature T of the system in Kelvins. The values that follow in the ligand-receptor matrix are theoretical differences in Gibbs free energies between a given bound and unbound ligand-receptor complex in kJ/mol.

$$\Delta G = G_{unbound} - G_{bound} \quad (1)$$

The unbound Gibbs free energy of the complex is 0 because it does not exist, so the negative Gibbs free energy of the resulting complex is the result. Our calculations that will be computed for each ligand-receptor pairing revolve around the Boltzmann distribution. The Boltzmann distribution is a concept in statistical mechanics that is used to predict the probability of observing a system in a state with the given energies of all of the possible states and the given temperature of the system. With all of the energies given for the possible conformations of the system of ligands and receptors, we can use apply the Boltzmann distribution to the energies to predict the probability of observing each specific ligand-receptor complex occurring.

$$p(C_x) = \frac{e^{-\Delta G_{C_x}/k_B T}}{\sum_{i=1}^N e^{-\Delta G_{C_i}/k_B T}} \quad (2)$$

In the above equation, the probability of observing a ligand-receptor complex C_x is the Boltzmann factor for C_x , which is exponential of the Gibbs free energy for C_x divided by the Boltzmann constant and temperature T of the system. The Boltzmann factor for C_x is divided by the sum of all of the Boltzmann factors for the N complexes that the respective ligand and receptor for the complex C_x are involved in, which is known as the partition function Z .

$$Z = \sum_i e^{-\beta E_i} \quad (3)$$

We are neglecting to use the concentrations in these calculations because we are assuming that the number of ligands and receptors involved is too large to keep track. Thus, the resulting probability values will be more like respective ratios for a complex's presence on a scale completely dependent on temperature and energetics.

The calculations have been implemented with success in Python scripts, but functions computing the calculations have not yet been transferred to the C code yet because the code structure it depends on (input and outputs) has just reached completion. The calculations are going to be implemented in three functions. The first calculation step is going to be a function that takes in the data array, iterates over every ΔG element, and replaces it with the Boltzmann factor form. The second function will iterate over the data array and sum the Boltzmann factors for every row, storing it in an array for the receptor partition functions, and then do the same for every column in an array for the ligand partition functions. The


third function will then iterate over the original array and divide every Boltzmann factor value by the sum of the respective ligand and receptor partition functions involved for the ligand and receptor pairing represented by the Boltzmann factor before finally replacing the element in the array with the probability value p . After the third calculation function is called, the array will be ready for output. How to implement these functions in a parallel format that will utilize the design of the Stampede cluster still needs to be researched.

iii. File write

Based on the Matrix Table, the program traverses the array and writes the output file into a .csv which will be used for data analysis, other scientific computation, and further research. Problems we ran into include understanding the proper syntax of a .csv file and the print routines that C provides, `printf()`. Once we found the proper print routine for file output, `fprintf()`, it was a matter of writing the .csv correctly. We ran into a problem where a hanging comma is written on each row before each new line clause is written. The problem with a hanging comma is that it will yield a null value which would cause errors in calculations and data visualizations. The programming error was quickly solved with an if statement. The current function is flexible where it can read a matrix of any size.

The .csv file could be readily used in data visualizations or statistical analysis programs such as Rstudio and Paraview. We hope to find interesting results such as similarities in the computed values and the data yielded from experimental trials.

iv. Make



```
stc-project/bin/:
test

stc-project/doc/:
projectproposal.pdf

stc-project/src/:
Makefile      ligand_receptor.c  lrfunctions.c      lrheader.h
```

Figure 2: *The structuring of the project directory.*

For the makefile, we wanted to make it flexible and easy to maintain incase we changed the structure of our project. We based the Makefile for the project off of the one given to us in the Make lecture. We used variables, functions such as wildcard and patsubst, and automatic variables to give us the flexibility we wanted. This makes it easy to adapt the makefile as we progress with our project. By using variables such as SRC and OBJ, as well as the automatic variables, when we add additional C files to our project, we will not need to make additional rules, just maybe have to adjust our old ones. The LDLIBS and LDFLAGS variables make it easier to modify the parts we need to if we decide we need to utilize any libraries. The CC variable is set to the intel (icc) compiler, as that is the one we are using in our project. The CFLAGS variable is currently set to and optimization level of 0 (-O0). When compiling, we only had one warning come up, indicating that the `lm` flag should not

be implemented before the files are set or it could affect performance. To remedy this we simply moved the flags to the end of the compile command.

Currently, our makefile builds one executable called `ligand`. This executable is dependent upon two object files, `lrfunctions.o`, and `ligand_receptor.o`. These object files are dependent upon our two C programs, `lrfunctions.c` and `ligand_receptor.c`, as well as the header file, `lrheader.h`. The header file will declare the functions defined in `lrfunctions.c`. The functions are then implemented in `ligand_receptor.c`. Lastly, we use phony rules in the makefile to clean up the directory. As we built more into our project, we will need to adjust the Makefile to account for what we add. When there are problems with compiling, we will add debugging options into the make file such as the `g` flag. We may decide to change the optimization flag from `O0` to something else. The way we structured the make file allows us to be very flexible in terms of our options, and will help us achieve a highly optimized program. We havent had to do much debugging, but when we need to we can turn on debug mode to help with the process.