A stochastic algorithm for quantifying partial solutions of the Drake Equation

G. Lentner

LL15 Natural Science Building, University of Louisville, Louisville, KY 40292 USA
email: geoffrey.lentner@louisville.edu

Abstract: Here I review a new code developed to analyze the quantitative effects of estimates on statistical profiles for the distribution of planets in galaxies as well as the number of planets distributed therein. This application (hereafter referred to as GAIA) models galaxies by interpolating N (number of) pseudo-random positions from arbitrary probability density functions in three dimensions given to it. GAIA takes these positions and does a *nearest neighbor* analysis to estimate the expected separations between bodies and solves for the sample standard deviation to establish a confidence interval for its results. Conceptually, the idea here is that we now have an understanding of how the early terms in the Drake equation are distributed spatially throughout the galaxy. If the trailing terms in the Drake equation (such as whether a civilization survives itself) cannot be modeled with a spacial profile because it is a function of local evolution and stochastic events in time that may continue to elude refinement; we can assume all possible end results, N, and distribute planets based on the profiles we know and study how the galaxy might look under these conditions. GAIA is a tool that, if used properly, can have implications for the Fermi Paradox, shedding light on how preferable a location we live in for detecting our neighbors. In this paper, I'll outline the code's construction and the user interface, as well as describe it's current issues and limitations. The code is available from its master online repository: `http://github.com/gLENTNER/ProjectGAIA.git`

Key words: statistical Drake Equation, Monte Carlo, Fermi Paradox, SETI, probability densities

1 Introduction

The prospect of detecting (or making contact with) an Extraterrestrial Intelligence (ETI) is supreme and its implications for our collective human perspective goes without saying. There are many ways in which an estimate on the total number of ETIs in the Milky Way can be expressed, the most prominent construction being the Drake Equation devised in 1961 by Frank Drake at the first meeting of SETI [1]. It was initially only intended as a thought tool to prompt intellectual curiosity; however, it is often used today as a serious method for constructing estimations. It has been revised [2] to be more explicit in its parameterization and revisited often. In my present discussion, I'm going to adopt the below formulation [3], which represents a typical construction considered today:

$$N = R \cdot f_p \cdot n_e \cdot f_L \cdot f_i \cdot f_c \cdot H_T \cdot H_* \tag{1.1}$$

where the parameters are defined in Table 1. The particular parameterization is not really what is important, but more so that we include all the spatially distributed qualities necessary for the later terms. I'll explain. I think it is the case that the components concerning evolution of life and intelligence are localized; that is, whether it occurs is dependent on scales of the solar system, not the galaxy. Furthermore, they are more difficult (relative to other terms in the equation) to model as of today because we simply don't have enough data. A component often included in current studies of this kind is a factor that deals with the potential for extinction events or destruction of the biosphere (potentially to later regenerate or otherwise lost forever). This could be potentially handled though by considering what is

Table 1: Table of Drake Equation Parameters.

R	Average star production rate.
f_g	Fraction of stars that are single F, G, or K dwarfs.
f_p	Fraction of stars with planets.
n_e	Number of suitable planets per star.
f_L	Fraction of suitable planets which evolve life.
f_i	Fraction of life bearing planets which develop intelligent life.
f_c	Fraction of planets with intelligent life which develop a technological civilization.
H_c	Characteristic time for evolution of a civilization.
H_*	Characteristic decay time for galactic star formation rate.

now being referred to as the Galactic Habitable Zone.

To restate it briefly: I am proposing for consideration a new treatment that simply models the distribution of relative probabilities of characteristics necessary for life to evolve and survive. We can posit a total number based on current best estimates of the discriminating factors for ETIs and then build Monte Carlo Realizations (MCRs) of the galaxy to study the impact of such a result. There is the potential here as well to take a backward look and place constraints on N. If we have a defined level of confidence in our statistical models for habitability in the galaxy, we can consider our confidence in the lack of ETIs within a local volume as a discriminant for a range of solutions that would predict a neighbor within that volume to the same level of confidence.

To this end, I've developed a new code (here after referred to as GAIA) that does just this. Given a set of probability density functions (PDFs) in any of three dimensions, it builds populations of a requested size that meet those statistics and performs a nearest neighbor analysis. The output is the expected separation between neighbors as a function of galactocentric radius and the sample standard deviation of that measure based on the number of trials (MCRs) constructed.

In the next section I'll give a brief overview of the types of profiles that might yield the best results; however, this is more so a description of the tool, and not my employment of it (which is forthcoming).

2 Probability and Statistics

This paper is not concerned with a lengthy discussion of probability theory and techniques in mixing PDFs and their resultant cumulative distribution functions (CDFs). With that said, I'll include a brief discussion of some of types of profiles I think are appropriate to include and how they are treated.

In the radial dimension, we can model the stellar number density with a mass density profile. One of the first requisites is that a planet be in orbit around a star, so we must go where the stars are. The surface density profile of the galaxy is generally speaking something of a decaying exponential.

$$\rho(r) \approx n_0 \exp\left(-r/R_D\right) \tag{2.1}$$

Additionally, the vertical structure of the galaxy can be modeled with a decaying exponential. This gives us planets distributed throughout the galaxy consistent with the stellar population. As for an angular profile, the spiral arms contain the denser star forming regions, and one might think to include something of this nature; however, F, G, and K type stars are not so bound to the spirals as they are much longer lived. This means that they are largely distributed isotropically in the disk.

We might also consider the metallicity gradient of the disk as terrestrial planets are relatively speaking more likely to be found around stars born of metal rich material. A radial profile that respects this is warranted.

Lastly, as I stated previously, there are regions where it is reasonable to think it is more/less likely that a biosphere would remain undisturbed in comparison to other regions. If a system is in a more dense region of the galaxy, it is more likely to have material thrown in to it's inner solar system. As well, supernova events are more of a threat. This has implications for life in the spiral arms. A system is at it's best chance (I'll hypothesize) when it is not passing in and out of the arms. So we might expect that there is a profile consisting of a tight gaussian-like curve around the radius of co-rotation. This is the radius at which the system would have a circular velocity equal to that of the pattern speed of the spiral arms. This would create a belt of habitability in the galaxy.

These types of considerations can be included in a set of profiles to be mixed, integrated, and used by GAIA to analyze what the implications would be for the expected neighbor separations between ETIs in that environment.

3 Code Structure

GAIA is programmed with an object oriented design patter. There is a hierarchical structure of ownership and management. In the following sections, I'll layout the components of the code and brief descriptions of their responsibilities. The structure is depicted graphically in Figure 1 at the top of the next page.

3.1 Tools

Throughout the simulation, there are two operations that happen with high frequency. That is the generation of sets of psuedorandom numbers and the act of resampling data onto a fixed grid. These algorithms already exist in the public domain in abundance. My goal was not to "reinvent the wheel" so to say.

In the case of the PRNG, I needed something very high performance with a long period. Most all of the scripting languages employ the exact same algorithm (mt19937). In my case, I not only wanted this for C++, but I needed it implemented as a *class* so I could spawn multiple independent threads for generating numbers in parallel. The new C++11 standard in fact has such an implementation, `std::mt19937_64`. As it turns out, my code produces identical output as the C++11 implementation. After I discovered it's existence I decided to just stick with mine because it was local and didn't require the use of experimental libraries.

As for the interpolation, I wanted an algorithm that was *in house*, fast, parallelized, and specialized for my purposes. I needed to know exactly what it was doing, and I needed it to do some extra house-keeping for me.

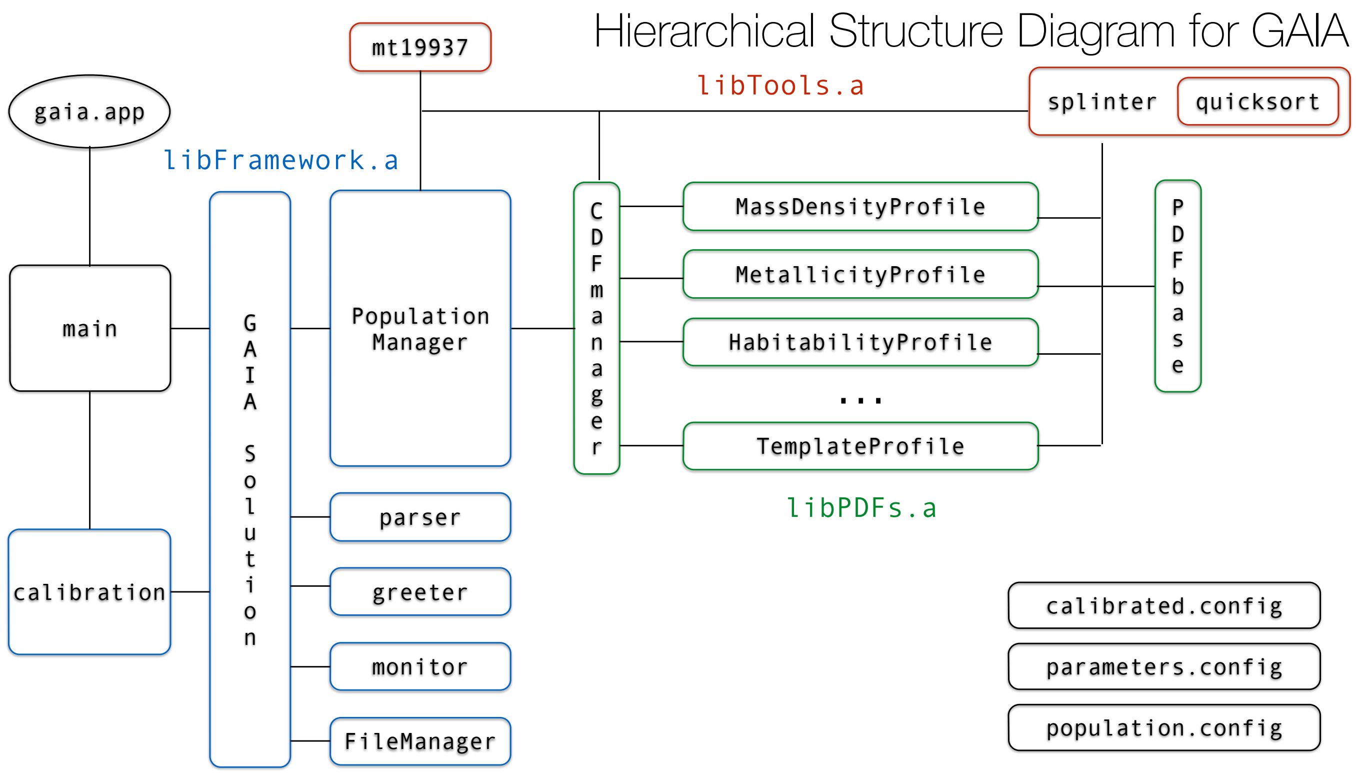

Figure 1: Above is a schematic diagram that shows how the GAIA application is constructed. In general, the scope flows down from left to right (with the exception of the parser, who is called by all the objects, but simply instantiated first by the application in GAIAsolution).

3.1.1 Pseudo-Random Numbers: mt19937

GAIA uses the Mersenne Twister as developed by Matsumoto and Nishimuro [4, 5]. The algorithm is a generalized feedback shift register. The code uses the 64-bit version of mt19937 as written by the developers themselves. The C-code was used to construct a C++ class and stripped to its essentials. As I stated previously, the object was necessary so that I could construct an array of independent instances (seeded separately). With this, I developed a routine (part of the `PopulationManager`) that constructs an array of pseudorandom numbers by have multiple generators working on delegated segments of the array in parallel.

3.1.2 SPLine INTERpolation

During the initialization process, the PDFs can optionally be read in as empirical data from an ascii file. If this is used, it is necessary to fix that data onto the working line-space for that dimension. This is for two reasons. First, there may be more profiles along that dimension, in which case it will be necessary to have the data at the same positions in order to properly combine them. Second, not only do we want to ensure the resolution of the data for the integration process, but GAIA uses midpoints a la Simpson's 3/8th's rule. Also, and more important, for every trial MCR that the code builds, we need to be able to compare the same radial values when averaging.

GAIA uses natural cubic spline interpolation to resample data. The code is my own, the algorithm is not. I simply referenced the nicely developed explanation available via *Wikipedia* [6, 7].

I wanted to have an object that constructs the spline polynomials upon being instantiated and retains these data. For obvious reasons, I didn't want GAIA to be building the polynomials every time it needed to interpolate. Furthermore, many implementations do not incorporate a sorting algorithm. Generally speaking this is not necessary because you would like to be able to interpolate around some function that might not be single-valued. As far as we are concerned however, we want our data in ascending order. The `splinter` object uses an adaptation of the *quicksort* algorithm. Quicksort is a recursive algorithm that partitions an array of values around a pivot value and then repeats on either side of the pivot. The largest of the arrays feed to `splinter` are the random numbers (which are uniformly distributed), so the pivot was taken as the midpoint, as there is no gain by trying to make guesses at an optimal pivot. The algorithm was adapted to work on two dimensional arrays (using the column as the sorting array) and the other columns strung along with it. Further, the function was overloaded with a wrapper that segments the array into intervals. The array is pre-sorted in parallel by having the recursive function operate on independent segments simultaneously. This gives a set of sorted pieces that then can be *zipped* back together in serial for the final sorted array. For sufficiently large arrays, significant speed ups can be had here.

Finally, the interpolation routine is a member function that returns a new array, given a new sample grid. By default, the function returns strait lines out of the ends of the data, sticking true to the *natural* in natural cubic spline interpolation. That is, the second derivative of the ends of the outside intervals is by definition zero. As a result, any new sample value outside the outermost intervals should get evaluated as a strait line off the first derivative of the last data point. However, it has as an option to return a null result. This is important for the resampling process between trial MCRs. The positions of the planets in the model are used as the data to construct new spline polynomials. When the fixed radial line-space is used to interpolate on those intervals, the outer intervals are very likely to be outside the data. Instead of returning poor results, we return zero and track the number of times this occurs on a given grid point. In this way, we can keep more accurate results only with larger uncertainty.

3.2 Framework

The essential body of the code is found in the application class `GAIAsolution`. This object is the primary owner of the many arms of the code, each responsible for a different set of tasks. The application is then wrapped in a main function as an executable. In the following sections I'll outline these different components and their responsibilities.

3.2.1 Main

The executable is compiled from `main.cc`. This code is simply a wrapper for the application. Although it does perform some tasks. The `monitor` is a singleton, and so it can be grabbed by the other objects during runtime. Main instantiates this object first because it keeps a set of clocks, one of them is the total simulation run time. As such, the very first thing that happens is the initialization of the monitor to start the simulation clock. Also, after the application object is created and initialized (with the arguments passed to main), we check to see if we are actually going to run the code or if we simply want to perform the auto-calibration routine. Main decides which of these two will happen. Lastly, if the simulation is being run in *verbose* mode, main displays the total time elapsed after the simulation is finished running. This is simply a top level function the puts everything together.

3.2.2 The application class

As stated previously, `GAIAsolution` is the primary application object. It owns or operates the majority of the essential arms of the code. Namely, it creates and uses the `parser`, the `FileManager`, the `monitor`, and the `PopulationManager`. It only has two functions, the initialization routine and the actual simulation function.

The initialization function takes the arguments passed from main and hands them off to the `parser`. After creating and initializing all the objects, it verifies that the initialization was a success. All of the objects maintain the same design pattern of initialization with a flag that signifies a failure. If any daughter object fails in its proper initialization, the parent object catches the failure and signals its own failure and tosses it up the line.

The simulation function is largely just a set of nested loops that iterates over all the population sets to be models. Each population is then iterated of the requested number of trials. For every trial iteration, a new population (MCR) must be built. Then the nearest neighbor analysis must be performed. After we solve for the separations, we need to resample this data onto a fixed grid and save it for later retrieval. The `PopulationManager` performs these tasks, but the application class delegates them.

3.2.3 Parsing input

When the code is executed, there is a moderate list of default parameters. All of these options can be changed, either by passing the argument at the command line or by specifying it in one of the configuration files. All of this input needs to be interpreted. The `parser` first takes the arguments that optionally were passed originally from main. It checks to see if they are valid and than swaps the default parameter with the new value.

Next, it reads in and interprets the *parameter file*, followed by the *population file*. If the `--auto-calibrate` flag was not given, it signals that we are going to run the code and reads in the *calibration file* that was created by the `calibration` object. This object is a *singleton*. In this way, it can parse the input data and simply hold onto it, waiting for the other objects to request specific pieces of information without the need to pass such information as arguments.

These processes are strait forward. The code for this object is the longest in the project merely because it contains an extensive list of error messages that it gives the user at every possible misstep. There are a lot of things that need to be in place for the code to run and the parser makes sure they are all in order, signaling the user what the particular problem was that halted the simulation before it started.

3.2.4 Managing file input/output

The `FileManager` is in charge or writing the output files and also reading in temporary files created by GAIA. When the simulation is initialized, the `FileManager` asks the `parser` for information about how many population sets will be run and what sizes. It then constructs the output file names with appropriate extensions for repeat population sizes. After all the trials are complete for a given population size and the analysis is finished, the `FileManager` writes the results to disk as an ascii file. The file is in a three column format. The first column is always the radial line-space provided in the parameter file. The second and third columns are the mean and sample standard deviation respectively. Because the resolution of the radial line-space and the number of trials for a given population size would have potentially required more active memory to maintain matrices of such size, in order to guard the simulation from memory errors, GAIA outputs the results for each MCR into a temporary binary file that it understands. After all the trials are complete, during the analysis procedure, all of these binary files are read from one at a time.

3.2.5 Monitoring progress

As was eluded to in the description of `main`, the `monitor` is responsible for maintaining a set of clocks. When it is first created, it starts the simulation runtime clock. This object also follows the singleton design patter, but this time only so that it doesn't lose the time. The primary responsibility for the `monitor` isn't about how long GAIA runs however, it is more important that it monitor progress. For any major use of the code, the program would likely be executed with the `--set-verbose=1` option (as oppose the to the default value of 2). This suppresses the following function entirely, because GAIA ought to be run on a server for an extended period of time in the background. However, for shorter runs, the `monitor` has a member function that displays a progress bar between trials. The progress function is more sophisticated than that though. Any run where the user would be actively monitoring the progress of the simulation is likely to be of a small population size. The iteration time on any population roughly $N \lesssim 10^3$ is quite fast. Simply putting a print statement inside the loop blindly however often results in the simulation being slowed merely because it is trying to print to the screen every time. With this in mind, the `monitor` keeps track of the frequency with which it's called upon and suppresses its own output, only refreshing the progress statement at a fixed frequency. By default this is half of one second.

In addition, the `monitor` includes with this progress statement the estimated date and time that a given run will be complete based on its current performance.

3.2.6 Population management

The real action takes place inside the `PopulationManager`. It is responsible for not only the construction of new MCRs but also the analysis. In order for it to generate new populations it needs a set of cumulative distribution functions. For this, it creates and initializes the `CDFmanager`. I'll discuss this process in the sections following this one. Once the CDFs are created (for each dimension on which it applies), the `PopulationManager` uses the `splinter` object to construct a set of spline polynomials on the CDF arrays.

Every time the build process is requested, the `PopulationManager` generates random number arrays and decides whether to use them to produce uniform distributions (on some domain) or if there is a CDF to interpolate. Depending on the population size and the information provided from the calibration file via the `parser`, the `PopulationManager` can create and initialize a family of parallel PRNGs to work on the arrays.

The nearest neighbor search is strait forward. As we have to know every distance to every other planet, there is no real opportunity for a sophisticated approach. The function that is in charge of this task though has two versions of itself. If the population size is small enough, we can manage a two dimensional array of size N^2 (not the case for most instances). If this is in fact the case (and the default threshold is $N = 10^4$), we need not double measure. The r_{ij} separation is the same as the r_{ji} separation. In this way, we can reduce the total computations necessary. Unfortunately, there isn't anything to be done for larger population sizes. However, this operation is parallelized using the OpenMP library (as are all of the other parallelizations in GAIA).

The final analysis routine reads in all the temporary binary files created by the `FileManager` after every iteration and co-adds them for a mean value and then computes the standard deviations.

3.2.7 CDF management

While all the other essential components of the code are necessary, the profiles that the user provides are the real science. In order for the `PopulationManager` to construct new MCRs it needs CDFs to interpolate onto. When the `CDFmanager` is created and initialized, it owns a list of *known* PDFs. By this I mean it has a list of all that have been defined for it. It asks the `parser` for which PDFs we want to use and it initializes them. When a particular PDF is initialized, it is subsequently put onto a separate list specific to the dimension (i.e., radial, angular, or vertical). In order to maintain such lists, all of the PDF class objects must be derived from the same base class. This is `PDFbase`, described in the next section.

After picking all the profiles and either evaluating them on the appropriate line-space or reading them in from a file and re-sampling, the `CDFmanager` multiplies all PDFs together along a given dimension. Typically, this means that the radial PDFs will be multiplied together.

After we have the master profile in all dimensions, we must integrate. GAIA uses Simpson's $3/8$ rule to solve the general integral. The `CDFmanager` uses `splinter` to interpolate a set of two midpoints between all the grid points and applies the rule on each interval to include a cumulative summation, resulting in the general integral.

The `CDFmanager` maintains these CDFs (or whether or not they even exist) and passes them off to the `PopulationManager`.

3.2.8 PDF base

The `PDFbase` class is an abstract class. It is composed of many *pure virtual* functions. It exists to facility the use of lists to maintain the profiles. It describes the essential structure of the derived PDFs. Originally, it was thought that this was the best way to go, having a nearly pure abstract class. However, it really only needs to be the case that function describing the analytical form of the PDF (if desired to have one) be virtual (such that it can be redefined). At present, all derived PDF class objects have a redefinition of the initialization functions (which are identical). This is not necessary, but currently there is no performance loss so it hasn't been changed. In future versions, this will be addressed for the sake of clarity and simplicity.

3.2.9 Derived PDF classes

The profile objects are derived from the `PDFbase` class. Every new profile needs to have a dimension specified at the very least. If your profiles are going to be read in from files, than little differs between the code of these objects apart from the names. If your profile is going to take on some analytical form, that must be specified. There is not much to say here except point to the following sections where I describe the user interface and how to create new profiles.

4 User Interface

4.1 Configuration and new profiles

The package comes with the following directory structure:

```
LICENSE
README.md
data/:

example/:
            calibration.config
            parameters.config
            population.config
src/:
            ...
tmp/:
```

The `LICENSE` file contains the GPL v2.0 license. `README.md` contains much of the same information being described here. The directories `data/` and `tmp/` are empty. They are the default names of the directories to which GAIA writes the final results and the temporary binary files respectively. Alternatives are discussed in the next section. The `example/` directory contains three configuration files. The names of these files are the default names that GAIA looks for if no alternative is specified. They are meant as an illustrative example of how to construct these files. The user can move these out of `example/` and alter them for their own purposes.

The `src/` directory contains all the source code. One can build GAIA simply by executing:

```
cd src/ && ./configure && make
```

This is not particularly helpful however because we haven't made any profiles yet. Let me start by saying that the `configure` script is not yet the conventional script users might think it is. It is simply a bash script with the same name. This script compiles three C++ programs and runs two of them. As was described earlier, the `CDFmanager` is responsible for maintaining a list of *known* PDFs. The convention here is that any file that ends in *Profile* is expected to be a derived PDF class. There are several pre-made options available in an example directory:

```
src/example/:
                 HabitabilityProfile.cc
                 HabitabilityProfile.hh
                 MassDensityProfile.cc
                 MassDensityProfile.hh
                 MetallicityProfile.cc
                 MetallicityProfile.hh
                 VerticalProfile.cc
                 VerticalProfile.hh
```

The user can move these out of their directory and use them immediately, or modify their analytical form by going into their source file. The more general option is to run the the configure script once and then use the automated tool provided with GAIA to make your profile for you. As an example:

```
 /ProjectGAIA/scr/:  ./configure
 /ProjectGAIA/scr/:  make profile
./makepdf
Name of new Profile:  myNew
Dimension for myNewProfile ([0],1,2):  1
Analytical form ([none]):  A * sin( omega * t )
What was the dependent variable ([r]):  t
List parameters:  A, omega
Value for [A]: 1.0/2.0
Value for [omega]:  3.1415926
./configure
```

This automated tool allows the user to construct new profile objects. Simply by answering these questions, the package automatically constructs the new source and header file, updates `CDFmanager.cc` to include it in the list of known PDFs, and also updates the `makefile` to include it in the build process. Every time the user creates a new profile, `make` will need to be executed again to update everything.

4.2 Input files

When GAIA runs, it looks for three input files. By default, if unspecified, it expects `parameters.config`, `population.config`, and `calibration.config` to exist in the current directory (with an exception on the last one).

The parameters file is where you tell GAIA what your line-spaces are and what profiles you want to include out of the known profiles in this run. You can write C-style comments with double slashes and leave any amount of blank lines. Furthermore, it doesn't matter what order you include your input. But lines must follow a specific format. First, line-spaces must be one of three: `radial_linespace`, `vertical_linespace`, or `angular_points`. For the angular line-space it is assumed to range over $0 - 2\pi$. For radial and vertical the format is as follows:

```
<type_linespace> :  <start> <end> <num_points>
```

[1]A note about units: it is up to the user to be consistent with units. The units used for the radial and vertical line-space should be the same and they will be the units the results are reported in.

So to model a disk out to 15 kpc[1] and have a half-parsec resolution:

```
radial_linespace :  0 15000 30001
```

The number of angular points is a single number not three. The value for the number of angular points as well as in place of the three numbers for the vertical line-space, the special keyword, `none`, can be substituted. In the case of the angular points it means we are isotropic and will distribute angles uniformly. In the case of the vertical line-space it means we have no vertical structure and only a flat disk. All three of these need to be provide and the radial line-space is demanded.

The profiles are specified with the keyword, `profile`, and their inclusion takes the following pattern:

```
profile :  <name of profile> <value>
```

where `<value>` should be either the path to the file for initializing the PDF or the special keyword, `linespace`, which means we aren't going to read that one from a file but evaluate it using it's analytical form along the line-space that corresponds to its dimension. For an example, see `ProjectGAIA/example/parameters.config`.

The population file is much simpler. It should be an ascii file containing two columns of numbers. Every row represents a different population set, where the first number is the size of the population and the second number is the quantity of trials to build. As an example:

```
512     1000
16384   128
```

says to build two population sets. We are to build 1000 MCRs of size 512 and then 128 MCRs of size 16384.

Finally, `calibration.config` should take a similar format to the population file, but have a set of 16 numbers for thread counts instead of a number for the trials. For example:

```
512      1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
16384    1 1 1 1 1 1 1 1 1 1 4 1 1 1 1 1
```

says for the same population sizes, we want to run everything in serial for sizes 512, but for the second population, 16384, we want to spawn four threads for the 11th task (which happens to be the nearest neighbor search). For *this* configuration file, GAIA has a calibration routine that will auto generate this file based on empirical measurements and the contents of your population file and the maximum number of threads the user stipulates it's allowed to use. This is at least how it will be for the next version for the code. The `calibration` object has not yet been implemented. Furthermore, in future editions of the code the `greeter` will be implemented which runs GAIA in an optional interactive mode and requests information from the user, only to then actually auto generate the population and parameter files.

4.3 Execution

With everything assumed to be default and the configuration files present and accounted for, one simple must execute: `./gaia.app`. The following in Table 2 is a list of the optional flags that can be provided at run time:

Table 2: Table of GAIA flags.

`--set-verbose`	=	`0 - silent`
		`1 - some output`
		`2 - progress bar`

`--output-directory`	=	`/path/to`
`--temp-directory`	=	`...`
`--parameter-file`	=	`...`
`--population-file`	=	`...`
`--calibration-file`	=	`...`
`--interactive-mode`		
`--auto-calibrate`		
`--max-threads`	=	`integer` $>$ `0`
`--force-threads`		`...`
`--size-threshold`	=	`integer` $>$ `3`
`--trial-sets`	=	`integer` $>$ `0`

The parameters do what they sound like. All the arguments must be passed with an equal sign followed by the value with no spaces with the exception of the interactive mode and auto calibration flag with should not be given with an equal sign. The size threshold argument is the cutoff point for the auto calibration routine to stop testing parallelization. By default it will go up to the largest size in the population file or until it reaches the maximum number of threads or until it reaches the maximum size for its data type, which ever is first. The trial sets argument says how many times to test the tasks during the auto calibration routine. By default it only tests it once.

5 Discussion

I have as of yet not had the opportunity to make a real run with the program, so I won't be including plots of output. I need to do more research on the functional form I might use for the profiles I suggested. Thus far I've tested the algorithm with merely the radial profile without vertical structure. The results show something reminiscent of a root function, with increasing noise with distance from the center of the galaxy. I expect with larger population sizes, not only for the separations to decrease (as they must) but the noise to diminish to some extent. I think with the inclusion of a habitability profile, there will be a dip in the function near the orbit of co-rotation. The behavior of this curve is not necessarily surprising in any way, but the actual value of the function at Earth's radius and the standard deviation in the expected separations is what is of interest. It won't be until I can provide valid input that I get valid output.

This code is a tool that I hope to continue to develop in the hopes that it is useful to me to do real meaningful work in the SETI field. As I stated earlier, it needs a more sophisticated treatment of the probability and statistics. Also, it could be useful in its own right to have it optionally output the actually positions of the bodies for a single trial as a tool for generating initial positions for galaxy N-body codes.

As for the experience of the project, it has provided many challenges for me to overcome and I think I've succeeded in most of them. I spent a month getting the spline interpolation working correctly with the quicksort algorithm. Typically, when I ran into a problem getting a design or logic to work the way I wanted, I would research how others had done it, then implement my own version of that technique and get it to work from there.

6 Acknowledgment

In addition to thanking Professor Brown for his patience and continuous help on this project, I'd like to acknowledge my colleagues in the Astrophysics lab for their numerous opinions and constructive feedback, particularly Brian Leist for encouraging me to pursue it in the first place.

As well, I think it's appropriate to acknowledge Dr. Duncan Forgan [2] with whom I've only had brief correspondence but whose papers were my only insight into what has been attempted regarding Monte Carlo methods applied to SETI in this way. I failed to cite his work during the brief introductory sections of this paper [8, 9]. I hope to get feedback from him and others working in this area.

References

[1] F. Drake, "Discussion of space science board," *National Academy of Sciences Conference on Extraterrestrial Intelligent Life*, 1961.

[2] I. Sclkovskii and C. Sagan *Intelligent Life in the Universe*, 1966.

[3] G. Steven, "The drake equation re-examined," *Q.Jl Royal Astronomical Society*, 1981.

[4] T. Nishimura, "Tables of 64-bit mersenne twisters," *ACM Transactions on Modeling and Computer Simulation 10*, pp. 348–357, 2000.

[5] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Transactions on Modeling and Computer Simulation 8*, pp. 3–30, 1998.

[6] "Spline interpolation," *Wikipedia, The Free Encyclopedia.* [Retrieved: December 12, 2014].

[7] "Tridiagonal matrix algorithm," *Wikipedia, The Free Encyclopedia.* [Retrieved: December 12, 2014].

[8] D. Forgan, "A numerical testbed for hypotheses of extraterrestrial life and intelligence," *International Journal of Astrobiology*, 2009.

[9] D. Forgan and K. Rice, "Numerical testbed of the rare earth hypothesis using monte carlo realization techniques," *International Journal of Astrobiology*, 2010.

[2] University of St. Andrews