HARPPI v1.0 User Manual

David W. Pearson, Ph.D.

July 29, 2016

Contents

Chapter 1

Introduction: Why another parameter-file parser?

I created the HumAn Readable Parameter-file ParsIng (HARPPI) library with the goal of having a the ability to parse parameters files without the code having to know what the parameters are ahead of time. This became desirable because every piece of code I was writing that took a parameter file to run required me setting up a custom data structure for that piece of code, and then re-writing a function which assigned the parameter file values to the data structure. This worked, but became cumbersome. I also wanted the ability to add comments to make the setup of parameter files a bit easier should some else want to run my code. By simply include an example parameter file that contained comments explaining each of the parameters, any reasonably intelligent person should be able to make edits and run the code with new parameters.

I was aware of using XML or .ini files for parameters, and that there were a number of libraries for C++ that could parse those files. However, I wanted the parameter files to be human readable, and to have an easy format. After all, the point of the files (for me at least) is to be opened by a human, have certain values changed by hand, and then have some code run again with the new values. This is something that I do frequently in my work. I have code which creates mock galaxy catalogs that are used for covariance matrix estimation in studying the large scale structure of the universe through two-point functions. The mock catalogs need to match the target survey, so the box size they are created in, the number of galaxies need, and numerous other values needed to be easily changeable.

I didn't find any of the other libraries out there to be particularly easy to use, and the parameter-file format was often confusing. As such, HARPPI was born. The library is incredibly light weight. It contains a single class, parameters, which has only a handful of public functions, but should have everything you need. In the parameter file, each parameter is defined with its type, a key, and the value in a format that should be familiar to anyone that has ever written C++ code:

```
double bias = 2.339 # This is the galaxy bias for the mock
```

The code doesn't care how much white space you have between the type, key, value and comment, as long as the first word is the type, the second is the key, followed by an equal sign and then the value, and obviously anything after the # is a comment. You can even have whole lines be a comment by starting them with a # followed by a space, then the comment. After you set up your parameter file, a built in function will parse it and store the values, which can then be accessed easily through other built in member functions.

You can simply set up the parameter file, read it, and get on with coding. No custom data structure for the set of parameters needed in that piece of code, no custom functions to read those parameters, and a simple to setup file format.

Chapter 2

Installing the Library

Currently, only linux/unix based operating systems are officially supported. As per the GPL, I cannot ensure that the library will work on your system. That said, there are no dependencies on anything outside of the C++ STL, so as long as your compiler supports c++14 standards, it should compile. As for getting it to be usable with your specific compiler on a specific system, you're on your own (for now at least). This library has been tested and is known to be working on Fedora GNU/Linux 23 and 24 with g++ version 5.3.1. If you are able to install and use the library on a different operating system with a different compiler please let me know[1]. Be sure to include any steps different than those outlined here.

There is a makefile provided with the library. Simply typing `make` will build it in your current directory. There are two other rules in the makefile that will place the library somewhere making it easy to access across your whole (linux) system, localinstall and globalinstall.

Typing `make localinstall` will build the library and output it to `/home/YOURUSERNAME/lib` and move the header file to `/home/YOURUSERNAME/include`. Once that completes to use the library, you'll need to open your `.bash_profile` file and make sure it looks like this

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
. ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/.local/bin:$HOME/bin
LIBRARY_PATH=$LIBRARY_PATH:$HOME/lib
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/lib
CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH:$HOME/include

export PATH
export LIBRARY_PATH
export LD_LIBRARY_PATH
export CPLUS_INCLUDE_PATH
```

After making the appropriate changes, log out of your current session and log back in to apply the changes. You should then be able to simply `#include <harppi.h>` in your code, and compile with

```
g++ -std=c++14 -lharppi {Other Options/Libraries} -o ExecutableName SoureName.cpp
```

[1]Email me at dpearson@phys.ksu.edu

This is the ***strongly*** recommended method of installation. All changes to the system should only effect your user, and therefore not mess up your whole system. This is particularly important if you are not the only user of your system. It should be fairly straightforward, and should you want to remove the library, simply deleting the `libharppi.a` file in `/home/YOURUSERNAME/lib` directory, and the `harppi.h` file from your `/home/YOURUSERNAME/include` directory should do the job. The changes in your `.bash_profile` could be undone, but should do no harm to leave (you may even find it convenient for your own uses).

Typing make globalinstall should install the library for everyone on your system. This command requires super user privileges to work as it writes files to the `/usr/include` and `/usr/lib` folders on your system. Installing this way shouldn't require altering your `.bash_profile` to use the library. However, as it makes **global** changes to the system, this should only be done if you are the only user of a system or its sole administrator. I'd recommend backing up all important files prior to running this command. I don't think it should cause any harm, but if it does, you've been warned and I am not responsible for the damage. In essence, the command should make a new directory for the library, `/usr/lib/harppi`, build the library there and then provide a link in the `/usr/lib` directory, while also copying `harppi.h` to `/usr/include`. Again, these actions shouldn't cause any harm, but I cannot guarantee that (your system may already have something called libharppi which may cause conflicts, though I don't think another libharppi exists, if you want to do a global install it would be wise to check first).

In future releases, I plan on distributing `libharppi.rpm` and `libharppi.deb` files to make the installation process much easier. However, at this time I'm still learning how to build rpm and deb packages.

Chapter 3

How to use HARPPI

HARPPI was designed to make using parameter files as simple as possible. What this means is that the setup of your parameters for a piece of code you are writing is done completely within the parameter file itself. Once that is done, a few simple commands allow you to load and access your parameters in your code. This chapter will detail how to use the library and hopefully clarify any limitations of the library as well.

3.1 The Parameter File

Since most of the work to use the library is involved in the setup of the parameter file, this seems a logical place to begin. The parameter file can be any plain text file. On Linux systems this means you can create a text file and save it with just about any extension you'd like. I tend to use something like, `File.params`, to let me know that it is indeed a parameter file, but you could use `.txt`, `.dat` or whatever you'd like so long as it's a plain text file. As we briefly touch on in the introduction, all parameters must be specified in the following manner

```
<type name> <key> = <value> # <Optional comment>
```

A function in the library will read in the parameter file line by line and expects things to be in that exact order including the equals sign. For people familiar with coding, this should be fairly standard and easy to remember. The type name can be any of the types listed in Table 3.1, which again should be familiar to those used to writing C++ code. This was done intentionally for that reason.

Once you have specified a parameter in the file, you may provide an optional comment by inserting white space after the value, the supplying a # followed by a space, and then what ever you want to say. As with most programming practices, such inline comments should be kept short. If you need to provide longer comments for some reason you can start a line in the parameter file with a # followed by a space, and then the whole line will be treated as a comment and ignored. This allows you to make comments to remind yourself what particular parameter file was for, or make some comment about a block of parameters.

So, let's say you are writing a program to read in a bunch of data files and then do some analysis of that data. You would likely need to specify the number of data files you have (an integer), the base of the file file names (a string), and the file extension (also a string). Let's also say that part of your analysis software wants to fit some model to each data file (or to the average of the data files), and you need to provide some initial guess as to the values of the parameters of your model a and b (both decimal numbers). A minimal parameter file for this situation could look like this:

```
string      filebase    =   DataRun
string      fileext     =   .dat
int         numFiles    =   1000
double      a           =   2.339
double      b           =   0.759
```

Table 3.1: List of all currently supported types along with a brief description. For those familiar with C++ data types, nothing here should come as a surprise.

Type Name	Description
string	Holds a string of alphanumeric characters
double	Holds double precision floating point numbers
int	Holds integer numbers
bool	Holds boolean types such as `true` or `false`
vector⟨double⟩	Holds a list of comma separated double precision floating point values (no spaces), useful when you have multiple similar objects and need to specify properties which may be slightly different
vector⟨int⟩	Holds a list of comma separated integer values (no spaces), useful for the reasons mentioned above.
vector⟨string⟩	Holds a list of comma separate string values (no spaces), in case you need to provide a list of files.

Note that the amount of white-space does not affect the library, so having the file formatted like above is no problem. This can help improve readability of the file and provides a clear separation between the type, key, and value.

While the above will work, if you were to open it again several months after writing it, you may not remember what your intent was, or which program you intended to use it for, so you may wish to add comments to the file to make things a little clearer.

```
# Parameters for ModelFit
# 7-28-2016
string      filebase    =   DataRun     # The base of all filenames
string      fileext     =   .dat        # The extension of all filenames
int         numFiles    =   1000        # The number of data files to process
double      a           =   2.339       # My initial guess for the value of a
double      b           =   0.759       # My initial guess for the value of b
```

Now you've provided comments at the top that tell you that these parameters are for your ModelFit program, and that you created the file on July 28, 2016. You have also provided brief descriptions of each parameter in the file (Note: for this example the commenting is likely overkill, but is done for illustrative purposes, I recommend following general coding guidelines for commenting and only commenting as much as needed where there may be some ambiguity).

So, you run your program, feeding it the name of the above parameter file and everything works fine, except you're getting really funky results for the fitting of the model to your data. You then decide that maybe your initial guesses are the source of the funkiness, and make some modifications to ModelFit so that it will try fitting with a variety of initial guesses if you ask it to. You could modify your parameter file so that it has a boolean to tell your program to try different initial guesses and provide lists of initial guesses for the parameters of the model in that case.

```
# Parameters for ModelFit
# 7-28-2016
string      filebase    =   DataRun     # The base of all filenames
string      fileext    =   .dat        # The extension of all filenames
int         numFiles   =   1000        # The number of data files to process
double      a          =   2.339       # My initial guess for the value of a
double      b          =   0.759       # My initial guess for the value of b

# EDIT: 8-3-2016
# The initial attempts to fit the model to the data provided some odd results.
# As such, the parameters below will allow the user of ModelFit to turn on
# the option to try different pairs of initial guesses for the model parameters.
bool             tryMulti    =   true        # Set true to try different pairs
                                             # set false to use only the above values
vector<double>   aList       =   2.339,1.895,6.752
vector<double>   bList       =   0.759,0.991,0.485
```

Again, you were able to provide comments to clarify that you made changes to your parameter file on August 3, 2016, and the reason for the changes. You also provided comments that tell the user what the boolean does when set to true or false. Note that the list of values for a and b are set with the `vector<double>` type, and that the values are separated only by a comma with no white space between them. This is because the library will break the line up at the commas and if there is any white space that will be included with the string that is then converted to a double[1]. The other vector types would be set up just as the ones shown above.

As you can see, the format is quite simple and very similar to what you would do if you were defining the variables inside your code. This hopefully makes it quite easy to use, and hopefully the above examples have shown how to properly setup your parameter files.

3.2 Using the Library

Now that you know how to setup a parameter file, you need to know how to use it in your code. Every effort has been made to ensure that utilizing the library is as simple as setting up the parameter file (maybe even simpler). So, let's take a look at what your code might look like if you wanted to use HARPPI. Figure 3.1 shows what the code for the example from the parameter file section above may look like, this would be your ModelFit.cpp file (obviously with a lot of code missing and FitModel being defined in `modelfitter.h`). This shows the basic interface with the library. Declare a `parameters` object, then use the member function `readParams` called with the parameter file name (here assumed to be passed at runtime via the command line). After calling `readParams` any parameters can be accessed using one of four member functions which are specific to the type of data you want returned, `getd` for double precision floating point numbers, `geti` for integer numbers, `gets` for strings, and `getb` for boolean types (i.e. it's the word get with the first letter of the expected return type).

All of the "get" functions are called in the same way, with a string that matches the key of the desired value in your parameter file, and optionally (if you have a vector type) the integer number corresponding to the desired element. The function prototypes are

```
double getd(std::string key, int element = 0);

int geti(std::string key, int element = 0);

std::string gets(std::string key, int element = 0);

bool getb(std::string key, int element = 0);
```

By setting them up in this manner, should you have a `vector<double>` you could access its members with

```
p.getd("aList", 1);
```

[1] I plan on changing this in the near future so that white space can be included.

```cpp
#include <iostream>
#include "modelfitter.h"
#include <harppi.h>

std::string filename(std::string filebase, int filenum, std::string fileext) {
        std::stringstream file;
        file << filebase << filenum << fileext;
        return file.str();
}

int main(int argc, char *argv[]) {
        parameters p;
        p.readParams(argv[1]);

        // Some code here

        for (int datafile = 0; datafile < p.geti("numFiles"); ++datafile) {
                std::string file = filename(p.gets("filebase"), datafile,
                                            p.gets("fileext"));
                FitModel(p.getd("a"), p.getd("b"), file);
        }

        // Some more code

        return 0;
}
```

Figure 3.1: Sample source code that uses libharppi. This demonstrates the minimal use case.

This function call would return the second value in the list of values provided in your parameter file. That is to say that like C++ arrays, it assumes a zero-indexed value, meaning the first item in the list is element 0. Because of the optional element parameter in the function call, the same functions can be used to return individual values or values that are a member of a list without needed even more functions (I'm currently exploring ways that I may be able to reduce things to a single "get" function call, though C++ does not make that easy).

As you may be able to see, when writing your code you are building in what the allowed parameters are, and what their keys must be, by using the strings in the function calls. However, this is the only setup required in the code on your part. The `readParams` function doesn't care what the parameters in the file are or what they are called. It is simply parsing the file and storing the values with the keys in the parameter file. A logical question is then, what happens if you request a parameter that is there? For that situation there are two options for you as the programmer. One is to do nothing, HARPPI will then throw and error telling you what the requested parameter was and whether it was not present or simply an incompatible type for the "get" function that was used[2]. The second option is to call the member function `check_min` with a vector of custom structs provided by the library called `typekey`. As the name implies, this struct has two members which are both strings, type and key. At the beginning of your code you can set up a vector with the parameter keys that you need to be present in the parameter file, and their associated types. Calling `check_min` will then make sure that the parameters are present and of the correct type. If not, it will throw an error with some information to help fix the parameter file.

Why the two options? Well, if you plan on distributing your code that uses this library, it would be prudent to have the check done at the beginning of the code run. These functions are pretty fast, so the user will know

[2]Any numeric type can be returned with either `getd` or `geti`, and will be cast as the associated return type. This was done intentionally in case the user needed to have a double cast as an integer or vice versa, but does mean caution must be used when calling for a numeric type. Make sure the "get" function has the appropriate return type for the situation you call it in.

Table 3.2: List of the public member functions of HARPPI with brief descriptions.

Function Name	Description
`void readParams(char *file)`	This function will read in the parameters from the specified file and store them in the parameters object from which it was called.
`void check_min(std::vector<typekey> minParams)`	This function will search for the parameters specified by the keys in the vector minParams and ensure that the type also matches.
`double getd(std::string key, int element = 0)`	Returns the request key as a double if it is a numeric type, otherwise it throws an error. If the key points to a vector, element will specify the vector member to return. If the requested key does not exist an error will be thrown.
`int geti(std::string key, int element = 0)`	Same as `getd`, but returns an integer as long the key points to a numeric type.
`bool getb(std::string key, int element = 0)`	Similar to the `getd` and `geti`, except it returns a boolean type. If the provide key points to anything other than a boolean, an error will be thrown.
`std::string gets(std::string key, int element = 0)`	Similar to the above get function, but returns a string type. At this time, if a numeric type key or boolean type key is provided an error will be thrown.

almost immediately if there is some parameter they didn't supply, or a typo in the key name. They can then fix the parameter file and run the code again. If you didn't do the check, and suppose your code may take some time to run (i.e. analyzing the 1000 data files may take 5 hours depending on their size and what you are doing with them), the user may walk away or even leave the code to run over night. If it takes some time to get to the portion of the code where the expected parameter is missing, the code would throw an error, but the user wouldn't be able to do anything about it until possibly the next day. This would likely upset the users of your code, as they would then have to start the run again, and wait even longer for the results. However, if you are the only one that is using your code, you're less likely to have a mistake in your parameter file, and if you do, you have only yourself to blame anyway, so you may decide that you'd rather not take the time to setup the vector of typekey's and instead just play it fast and loose. Thus, the two options.

I do strongly recommend implementing `check_min` for any code you right that may someday be run by another person, especially if they don't have an example parameter file or documentation telling them what parameters are needed. To use `check_min`, first you have to setup your vector, then call the function. Figure 3.2 shows how the first example would change if you wanted to check for the minimum parameters before the code goes too far.

Table 3.2 lists all the member functions in a single place with brief descriptions as a handy reference.

```
#include <iostream>
#include <vector>
#include "modelfitter.h"
#include <harppi.h>

std::string filename(std::string filebase, int filenum, std::string fileext) {
        std::stringstream file;
        file << filebase << filenum << fileext;
        return file.str();
}

int main(int argc, char *argv[]) {
        std::vector<typekey> minParams(5);
        minParams[0] = {"int", "numFiles"};
        minParams[1] = {"string", "filebase"};
        minParams[2] = {"string", "fileext"};
        minParams[3] = {"double", "a"};
        minParams[4] = {"double", "b"};

        parameters p;
        p.readParams(argv[1]);
        p.check_min(minParams);

        // Some code here

        for (int datafile = 0; datafile < p.geti("numFiles"); ++datafile) {
                std::string file = filename(p.gets("filebase"), datafile,
                                            p.gets("fileext"));
                FitModel(p.getd("a"), p.getd("b"), file);
        }

        // Some more code

        return 0;
}
```

Figure 3.2: Sample source code that uses libharppi. This demonstrates how to check that certain parameters are present before the code proceeds to far.