LogNormal mocK aNisotropic cataLOGS (LNKNLOGS) version 1.0 User Guide

David W. Pearson

June 23, 2017

1 Introduction

This guide will give you a brief overview on using the LogNormal mocK aNisotropic cataLOGS (LNKNLOGS) software. This version is designed to create lognormal mock catalogs with anisotropies in a single Cartesian coordinate direction – specifically the $\boldsymbol{x}$ direction. This is done by reading in an input *matter power spectrum* and, when distributing it to the grid for the initial step in the lognormal procedure, multiplying by $(b+\mu^2 f)^2$ where b is the desired galaxy bias, f is the linear growth factor and $\mu = k_x/k$.

2 Installation

2.1 Dependencies

Most of what the code needs to do its job is included. However, there are a few external dependencies that need to be resolved prior to being able to successfully compile the code. The two external libraries needed are fairly common – the Fastest Fourier Transform in the West (FFTW) library version 3.3.3 or later and the GNU Scientific Library (GSL) version 1.15 or later. Once those are installed, the included makefile can be used to compile the code. Note that if either FFTW or GSL cannot be linked simply with

```
g++ -lfftw3 -lgsl ...
```

then you will need to modify the makefile so that those libraries can be linked. Alternatively, you could edit your `.bash_profile` so that it has the following lines

```
LIBRARY_PATH=$LIBRARY_PATH:{PATH TO FFTW}:{PATH TO GSL}
export LIBRARY_PATH
```

Additionally, this code uses features of the C++11 standard, namely the newly improved pseudo-random number generators and distribution. This means that your compiler needs to support those standards. It is recommended that you use the GNU Compiler Collection's (GCC) C++ compiler, `g++`, version 4.8.5 or later.

The code has been tested on Fedora 25 GNU/Linux and Scientific Linux 7, and functions on both those distributions. However, there is no reason to suspect that it will not function on other Linux distributions that have the minimum library versions installed.

To compile, simply navigate to the directory where you downloaded the code, unzip the archive, then type

```
$ cd LNKNLogs
$ make
```

into a terminal window.

3 The Parameter File

Include in the zipped archive is a sample parameter file named `LNKNLogs.params`. There is nothing special about the file extension, its simply a way of helping keep track of what the file is used for. This parameter file is parsed by a custom library called the HumAn Readable Parameter Parsing lIbrary (HARPPI) which I developed as a convenience to myself. HARPPI has several useful features starting with the fact that the setting up the parameter file itself is all that is needed for the code to do its job. You can note that in `main.cpp` there are two lines near the beginning

```
parameters p(argv[1]);
p.print();
```

The first of these passes the parameter file name when initializing the parameters object, `p`. That object then stores all the parameters which are then accessible through four member functions, `gets("PARAMETER NAME")`, `geti("PARAMETER NAME")`, `getd("PARAMETER NAME")`, `getb("PARAMETER NAME")` for strings, integers, doubles, and booleans, respectively. Because of the way this works, it is then important to ensure that all the needed parameters for the code are in the parameter file and named as expected in the code. If there is a missing parameter when you run LNKNLogs it will throw an error and stop the code, telling you which parameter is missing.

The provided example shows how these files should be structured. Lines that begin with `#` followed by a space are comments. Comments can also be included after setting a parameter, e.g.

```
int start_num = 1 # This is the number for the first mock
```

As you can see, declaring a parameter follows a C/C++-like syntax. For LNKNLogs you must provide the following parameters

- `int Nx, int Ny, int Nz`
  These specify the number of grid points in each Cartesian coordinate direction. In principle these can be set to any values, though FFTW works the fastest with powers of 2. **NOTE:** Very large grid sizes will cause LNKNLogs to consume large amounts of memory. For example, for a 512^3 grid it will need about 3 GB of memory, and for 1024^3 this increases to about 24 GB. To reduced memory usage, the code is being modified to use in-place transforms, but this is a work in progress.

- `double Lx, double Ly, double Lz`
  These specify the physical dimensions of the rectangular prism for the mocks.

- `double b`
  This is the desired galaxy bias.

- `double f`
  This is the desired linear growth factor.

- `double nbar`
  This specifies the desired number density of tracers. The output file size is sensitive to the number density and the volume of the mock. Also, a large number density may cause the code to run slower as it takes longer to write the galaxies to the file.

- `string mock_base`
  Specifies the first part of the file name to be used when generating the catalogs, this will be followed by the specific mock number and then the specified extension.

- `string mock_ext`
  Specifies the file extension to use for the mock catalogs. The code automatically inserts the '.' so you should omit that here.

- `int start_num`
  This tells the code what number the first catalog should have. The first time you run the code, you should set this to 1. For subsequent runs to get more catalogs, set this to 1 + the last numbered mock created.

- `int num_mocks`
  Specifies the number of mocks to create in a particular run of the code. The code loops from `start_num` to `start_num + num_mocks`.

- `int digits`
  Sets the width of the file numbers. If this is set to 4, the files will be of the form `LNKNLogs_0001.dat`.

- `string wisdom_file`
  Specifies where to store the wisdom accumulated by FFTW. On the first run of the code with a particular grid size (e.g. values of `Nx, Ny, Nz`), FFTW will need to create a plan to optimize the Fourier transforms. Once created, subsequent runs with the same grid size can simply use the information in this file to virtually eliminate the Fourier transform plan creation overhead.

- `string in_pk_file`
  Name of the file that contains the input power spectrum. This can be any plan text file in two column format, where columns are denoted by white space (tabs or spaces). The first column should be the frequency, the second column should be the *matter power spectrum* associated with those frequencies. See the include CAMB power spectrum for an example. **NOTE:** You will want your input power spectrum to go to a high enough frequency that all grid points have k-values lower then the highest frequency in your input file. Also, it is a good idea to insert a value of zero for the zero frequency to ensure the smallest frequencies are within the range. If $k_{f,i} = 2\pi/L_i$ (where i specifies one of the Cartesian coordinate directions) is smaller than the smallest frequency in the input file, GSL will throw and error.

- `bool max_threads`
  LNKNLOGS uses multi-threaded Fourier transforms in order to increase computation efficiency. This is done through openMP, meaning that the code is designed to run on a single multi-core processor. This parameter tells the code to utilize all of the available CPU cores if set to `true`. If set to `false`, the value in `num_threads` will be used. If you are uncertain how many threads your computer can run, leave this set to `true`.

- `int num_threads`
  Specifies the number of CPU threads to use for the parallel Fourier transforms. **NOTE:** Do not set this value to a number larger than the number of threads your CPU can handle. Doing so may reduce performance and cause inaccurate results. If you are unsure the maximum number of threads you CPU can handle, it is recommended to leave `max_threads` set to `true`.

4 Notes on measuring the power spectrum from these mocks

When using these mocks, there are corrections that you need to apply, or modifications to the expected power spectrum to perform. This sections outlines these corrections.

First, do not use a grid size smaller than the one used in the generation of the mock when binning the tracers. The tracers are placed in a uniform random manner within a particular grid cell. Using a finer grid to measure the power spectrum may impact the small scale power.

Second, since we use a discrete grid to do the spherical averaging when measuring the power spectrum, the expected values will differ from the what you would get by integrating the input power spectrum against the Legendre multipoles. To determine the size of this effect, distribute the input power spectrum, $\delta(\boldsymbol{k}) = ((b + \mu^2 f)^2 P_m(k)/2)^{1/2} + \mathrm{i}((b + \mu^2 f)^2 P_m(k)/2)^{1/2}$, to a grid and bin as if you were measuring the power spectrum multipoles from a mock catalog. You can either take the difference of these values with those from the integral and adjust your measured values, or simply use the result as your expected power spectrum.

Third, for the same reason as above, you do need to consider shot noise for the higher order multipoles, since the sum of discrete points can differ from the integrals.