User Guide for Corrfunc

Manodeep Sinha,
Department of Physics & Astronomy,
Vanderbilt University,
Nashville, TN 37235.
manodeep@gmail.com

January 26, 2015

Contents

1 Introduction

Correlation functions are a statistical measure of a density field and are widely used in large-scale structure formation. Generally, the measurements are done *once*[1] on survey data and compared with model predictions in a Monte-Carlo Markov Chain. As such, the correlation functions have to be measured repeatedly during an MCMC. The codes presented here are meant to cover the typical scenarios of measuring correlation functions in theory-land. The primary consideration in writing these codes is speed[2]– the codes presented here should outperform any other CPU based correlation functions codes by a wide-margin.

[1] which is why I have not bothered with releasing the codes to measure correlation functions on data

[2] The secondary consideration was maintainability and ease of use for others. I have versions of these codes that are even faster but are potentially gibberish for anyone other than me!

2 Installation

The only requirements for the code to install is a valid C compiler, with OpenMP support. The AVX instruction set can only be used for CPU's later than 2011 (Intel Sandy Bridge/ AMD Bulldozer or later).

2.1 Getting the Source

You can obtain the source in two ways: i) Clone the mercurial repo (`hg clone https://bitbucket.org/manodeep/corrfunc/`) or ii) Download the tar archive (corrfunc.$MAJOR.0.$MINOR.tar.gz) and unpack it in the directory where you wish to keep the files (`tar xvzf corrfunc.$MAJOR.0.$MINOR.tar.gz`). Here, $MAJOR and $MINOR refer to the major and minor release versions (current $MAJOR=1, $MINOR=0). I will only change the $MAJOR version if the API breaks.
The directory structure for the code looks like this:

```
corrfunc
├── paper
└── xi_theory
    ├── benchmarks........IDL scripts to run benchmarks.
    ├── bin..............Will be created to copy executable
    │                    files when you run 'make install'.
    ├── examples.........Source files for example C bindings
    │                    using the static libraries.
    ├── include..........Header files for static libraries.
    ├── io...............Source files for reading in data.
    ├── lib..............Will be created to copy static
    │                    libraries and python library after
    │                    you run 'make'.
    ├── python_bindings...Source files to generate python
    │                    bindings.
    ├── tests............Correct outputs for tests.
    │   └── data.........Mock galaxy catalogs for tests.
    ├── utils............Source files for creating 3-D grid
    │                    and helper routines.
    ├── wp...............Source files for w_p(r_p).
    ├── xi_of_r..........Source files for ξ(r).
    └── xi_rp_pi.........Source files for ξ(r_p, π).
```

Option Type	Option Name	Default State	Requires	Notes
Science	PERIODIC	Enabled	None	Enables periodic boundary conditions.
	OUTPUT_RPAVG	Disabled	DOUBLE_PREC	Outputs the average pair-separation in each bin. $\xi(r)$ and $w_p(r_p)$ can be slower by more than 2×, $\xi(r_p, \pi)$ is less affected.
Code	DOUBLE_PREC	Disabled	None	Computations are done using double precision. Slower and requires more RAM.
	USE_AVX	Enabled	CPU and compiler with AVX support	CPUs later than 2011 have AVX support. Code will run much faster with this option.
	USE_OMP	Enabled	OpenMP capable compiler	Since `clang` does not support OpenMP yet, `common.mk` will stop compilation with `clang` when this flag is enabled.

Table 1: List of compilations options, what the options mean and their dependencies for the codes.

2.2 Code Options

There are a few code options that control both the Science case and the code compilation. All of these options are located in 'common.mk' in the base directory ('corrfunc'). Edit the first few lines to set these options (see Table. 1 for details) :

- Science options – PERIODIC, OUTPUT_RPAVG
- Code options – DOUBLE_PREC, USE_AVX and USE_OMP

Depending on your Science use-case and the cpu/compiler, you will want to set the different options. Once you set those options, you should set the C compiler, CC (available options are `icc`, `gcc`, `clang`). Once you have set the compiler, installing should be as simple as typing 'make' and 'make install' in the `xi_theory` directory. All the libraries are intentionally chosen to be `static` libraries just to avoid any path conflicts. However, on MAC OSX, you may have to do more to get the library to work – so I have outlined some of the scenarios in Section 2.4.

2.3 Linux

If the installation went well, you should have an executable called `run_correlations` in the `examples` directory. Type `./run_correlations` in the `examples` directory and you should see the code in action. The C source file `run_correlations.c` also serves as an example to use the $\xi(r)$, $\xi(r_p, \pi)$ and $w_p(r_p)$ libraries in C.

2.4 Mac OSX

There can be two issues on MACs. One is that the default `gcc` assembler supplied by `XCode` or `macports` is too old and does not support AVX instructions even when the CPU does. One way to get around this is by using the `clang` assembler even when compiling with `gcc`. The easiest way to do it is by replacing the default assembler with the `as` script in the `paper` directory (taken from this url). Copy this `as` script to the appropriate directory (`/opt/local/bin/` for me since I use `macports gcc` on my laptop).
Another problem might come with running the python example codes in the `python_bindings` directory. If you get an error message:

- `Fatal Python error: PyThreadState_Get: no current thread`

when you run `python call_correlation_functions.py`, then the following steps might fix the problem (these are also noted in the FAQ). This error occurs when the python library used at compile time is not the same as the runtime python library. In all cases that I have seen, this error occurs when using the `conda` package manager for python[3].

- Change the relative path for the shared python library _countpairs.so. You can change the relative path by issuing the command:
  ```
  install_name_tool -change libpython2.7.dylib `python-config --prefix`/lib/libpython2.7.dylib
  _countpairs.so
  ```
- Add to the fallback library path environment variable.
  ```
  export DYLD_FALLBACK_LIBRARY_PATH=`python-config --prefix`/lib:$DYLD_FALLBACK_LIBRARY_PATH
  ```
- If both of the above methods fail, then create a symbolic link
  ```
  ln -s `python-config --prefix`/lib/libpython2.7.dylib
  ```

If all went well, then you should be able to run the `run_correlations` code in the `examples` directory as well as execute `python call_correlation_functions.py` in the `python_bindings` directory. In all of the above examples, I have assumed that the relevant python library is `libpython2.7.dylib` (the default under `conda`) – you may have to replace it with your python library version.

[3] This behaviour is by design according to `conda`

3 Running the Codes

3.1 Input Files

The codes currently can handle these types of input data files:

- `ascii` – White-space separated columns, format code is 'a'.
- `csv` – Comma-separated values, format code is 'c'.
- `fast-food` – Fast-food, fortran binary format, format code is 'f'. The fast-food file format is described in detail in Section 3.1.1.

For the `ascii` and `csv` files, the code reads in the first three columns as the co-moving `X/Y/Z` arrays. Note, that more columns can be present but the code will ignore those columns.

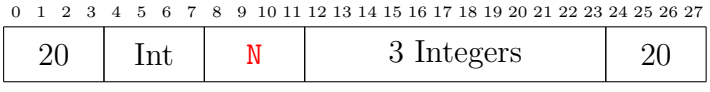
3.1.1 The fast-food file format

The fast-food format is a fortran binary format – all fields are surrounded with 4 bytes padding. These value of these padding bytes is the number of bytes of data contained in between the padding bytes. For example, to write out `20 bytes of data` in a fast-food file format would require a total of $4 + 20 + 4 = 28$ bytes. The first and last 4 bytes of the file will contain the value 20 – showing that 20 bytes of real data are contained in between the two paddings.

The `fast-food` file consists of a header:

```
int idat[5];
float fdat[9];
float znow;
```

For the purposes of these correlation function codes, the only useful quantity is `idat[1]` which contains `N` – the number of particles in the data file.

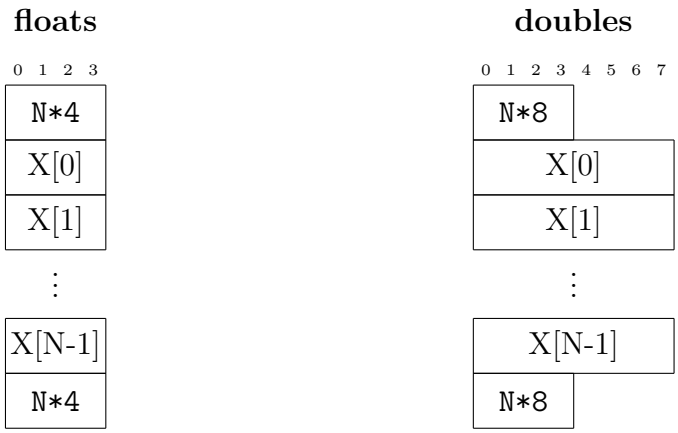


The next 56 bytes contain two other fields (10 floats + 4 padding bytes of 4 bytes each) and their corresponding paddings. Since these bytes do not

contain any data that are useful in the context of these codes we will skip over the contents of these bytes. In `io.c`, I actually `fseek` over these bytes.

After this header, the actual `X/Y/Z` values are stored. The first 4 bytes after the header contains `N*4, for float precision` or `N*8, for double precision` where `N=idat[1]`, is the number of particles in the file. After all the `X` values there will be another 4 bytes containing `N*4` or `N*8`. *Note, that even when the `X/Y/Z` arrays are written out in double-precision, the padding is still 4 bytes.*

Byte-structure of the X/Y/Z arrays in a fast-food file



3.2 Specifying the radial bins

The codes were intentionaly designed to read in a set of (somewhat) arbitrary[4] set of bin specifications from a file. This way, you can specify disjoint bin-edges as well as use 0.0 as a bin edge (which would be impossible if log bins are assumed). The bins are to be specified in a (white-space separated) text file in this manner:

```
r_low[0]  r_high[0]
r_low[1]  r_high[1]
.
.
.
r_low[nbins-1]  r_high[nbins-1]
```

[4] I assume bins are non-overlapping

where, `r_low[i]` and `r_high[i]` are the left and right edges of the `i`'th bin respectively. The text files should contains as many lines as the number of bins desired. The `logbins` executable can be used to create such a text file containing log bins. The syntax for running `logbins` is:

```
./logbins <rmin> <rmax> <nbins>   > <text filename>
```

An example of such a file with radial bins is the file `bins` in the `tests` directory.
Note, all of the three codes print the correlation function to ***`stdout`*** *– so be sure to redirect* ***`stdout`*** *to an output file.*

3.3 Running $\xi(r)$

To run the correlation function in 3-D, you will need to run the executable `DD` from either the `bin` or `xi_of_r` directory. The inputs to `DD` are:

- `file1` – the file name for the first file.
- `format1` – the file format for the first file.
- `file2` – the file name for the second file.
- `format2` – the file format for the second file.
- `binfile` – the file name for the file containing the bins (see Section 3.2)
- `nthreads` – the number of OpenMP threads to use (only required when the Makefile option USE_OMP is enabled)

Thus, the code can compute an auto-correlation (when `file1,format1` and `file2,format2` are identical) or a cross-correlation (when `file1` and `file2` are different). Out of the box, some sample `DD` call directory) can be:

- auto-correlation – `./DD ../tests/data/gals_Mr19.ff f ../tests/data/gals_Mr19.ff f ../tests/bins 4 > Mr19_output.DD`
- cross-correlation – `./DD ../tests/data/cmassmock_Zspace.ff f ../tests/data/random_Zspace.ff f ../tests/bins 4 > cmass_output.DR`

The output (printed to stdout) has `nbins` columns `<Npairs> <rpavg> <rmin> <rmax>`, where the `<rpavg>` column contains `0.0` unless the Makefile option OUTPUT_RPAVG has been enabled. Here, each line of the output represents the `i`'th radial bin.

3.4 Running $\xi(r_p, \pi)$

To run the correlation function in 2-D for $\xi(r_p, \pi)$, you will need to run the executable `DDrppi` from either the `bin` or `xi_rp_pi` directory. The inputs to `DDrppi` are:

- `file1` – the file name for the first file.
- `format1` – the file format for the first file.
- `file2` – the file name for the second file.
- `format2` – the file format for the second file.
- `binfile` – the file name for the file containing the bins (see Section 3.2)
- `pimax` – the maximum distance to consider in the π direction. The code uses `Z` axis as the π (line-of-sight) direction.
- `nthreads` – the number of OpenMP threads to use (only required when the Makefile option USE_OMP is enabled)

Thus, the code can compute an auto-correlation (when `file1,format1` and `file2,format2` are identical) or a cross-correlation (when `file1` and `file2` are different). Out of the box, some sample `DDrppi` call directory) can be:

- auto-correlation – `./DDrppi ../tests/data/gals_Mr19.ff f ../tests/data/gals_Mr19.ff f ../tests/bins 40.0 4 > Mr19_output_rppi.DD`
- cross-correlation – `./DDrppi ../tests/data/cmassmock_Zspace.ff f ../tests/data/random_Zspace.ff f ../tests/bins 80.0 4 > cmass_output_rppi.DR`

The output (printed to stdout) has `nbins` columns `<Npairs> <rpavg> <log(rmin)> <pi_upper>`, where the `<rpavg>` column contains `0.0` unless the Makefile option OUTPUT_RPAVG has been enabled. The code bins in 1 Mpc/h in the π direction by default. Therefore, the total number of bins in the output file will be $nbins \times \pi_{\rm max}$, where each radial bin is further broken into $\pi_{\rm max}$ bins along the π direction.
There is a code `wprp` in the `xi_rp_pi` directory that can combine the output of `DDrppi` for `DD`, `DR` and `RR` counts and use Landy-Szalay estimator to produce a projected correlation function.

3.5 Running $w_p(r_p)$

To run the projected correlation function, you will need to run the executable `wp` from either the `bin` or `wp` directory. The inputs to `wp` are:

- `boxsize` – the boxsize for the periodic box.
- `file` – file name.
- `format` – file format.
- `binfile` – the file name for the file containing the bins (see Section 3.2)
- `pimax` – the maximum distance to consider in the π direction. The code uses `Z` axis as the π (line-of-sight) direction.
- `nthreads` – the number of OpenMP threads to use (only required when the Makefile option USE_OMP is enabled)

The `wp` code only computes an auto-correlation with PERIODIC boundary conditions, irrespective of the options set in `common.mk`.

- `./wp 420.0 ../tests/data/gals_Mr19.ff f ../tests/bins 40.0 4 > Mr19_output.wp`

The output (printed to stdout) has `nbins` columns `<wp> <rpavg> <rmin> <rmax> <Npairs>`, where the `<rpavg>` column contains `0.0` unless the Makefile option OUTPUT_RPAVG has been enabled.

4 Code Design

The following sections are taken directly from the associated paper. The fundamental idea behind the code can be broken down into the following steps:

- Given the max. separation, r_{max}, for the correlation function, grid the entire the particle distribution with cell width r_{max}. In case of $\xi(r_p, \pi)$ and $w_p(r_p)$, the `X/Y` bins are r_{max} while the `Z` bins is $\pi_{\rm max}$.
- In each 3-D cell, store all the particles that are located in that cell in contiguous `X/Y/Z` arrays. This is contained in a structure defined in 4.1.
- Loop over all particles in every cell (this loop uses OpenMP parallelization if USE_OMP is enabled). This is the `index1` loop in `countpairs.c, countpairs_rp_pi.c and countpairs_wp.c`. The loop variable for the loop over each particle is `i`.
- Now, for every particle in the input list, we only need to look at particles in the neighbouring cells (see Fig. 1).
- Loop over neighbouring cells in all 3 dimensions. Corresponding variables are `iiix, iiiy, iiiz`. These variables are constructed from the loop variables `iix, iiy, iiz` to account for periodic boundary conditions. [5]
- Once we have a triplet of `iiix, iiiy, iiiz`, we can construct the index for the neighbouring cell. Variable `index2` gives the index for the neighbouring cell.
- Compute the distances between target particle and some particle bunch (where a chunk is 4 double or 8 floats) using AVX intrinsics. The slower, non-AVX version of the code computes the distances serially and does not take advantage of the CPU architecture. This is the loop with loop variable `j` in `countpairs.c, countpairs_rp_pi.c, countpairs_wp.c`.

[5] I apologize for this atrocious variable naming scheme.

- Check if any of the distances are less than r_{max}. If not, continue processing the data. If yes, then update the histograms using AVX `bit-masks` and `popcount`.
- Continue until all particles in neighbouring cell are done.

4.1 How to Maintain Cache Locality within the Grid

For all pairs around a given target galaxy, we need to compute distances to all points within all neighbouring 3-d cells. We ensure that the particle locations are contiguous by moving them into the following `C struct` in the order in which they arrive.

```
typedef struct{
  DOUBLE *x;
  DOUBLE *y;
  DOUBLE *z;
  int64_t nelements;
} cellarray;
```

Listing 4.1: Definition of the cellarray structure. This structure contains the `X/Y/Z` positions of all the particles that are in one 3-D cell.

5 Calling the C Libraries

All of the correlation function codes create a corresponding `static` library rather than a `dynamic/shared` library. This was a design decision intended to minimize path-issues for the end-user. After the libraries have been created, it is fairly straightforward to use them in an external `C/python` code.

5.1 C bindings

The `examples` contains the files `run_correlations.c` that shows how to use the three types of correlation function libraries from C. Essentially, the process consists of including the appropriate header file and passing the appropriate arrays in the functions.

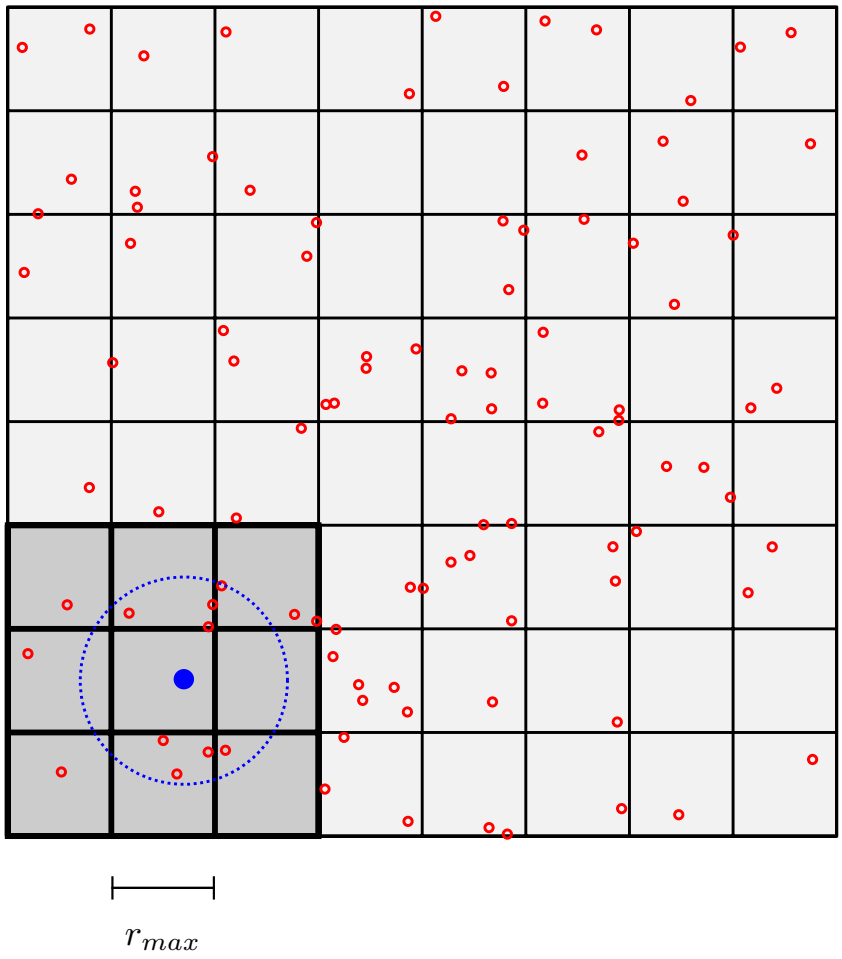


Figure 1: A 2-D grid showing the bin-lattice partitioning scheme. The bigger square show the entire domain, the red circles show a random distribution of 100 particles. Let's say we want to compute all pairs for the target blue point, then we would only have to consider red points that are within one cell (the dark shaded region). A circle with radius r_{max} is also drawn to shown the actual pairs that will eventually count in the correlation function.

5.1.1 API for $\xi(r)$

```
typedef struct{
  uint64_t *npairs;
  DOUBLE *rupp;
  DOUBLE *rpavg;
  int nbin;
} results_countpairs;

results_countpairs * countpairs(
const int64_t ND1, const DOUBLE * const X1, const DOUBLE * const Y1, const
    DOUBLE  * const Z1,
const int64_t ND2, const DOUBLE * const X2, const DOUBLE * const Y2, const
    DOUBLE  * const Z2,
#ifdef USE_OMP
const int numthreads,
#endif
const int autocorr,
const char *binfile) __attribute__((warn_unused_result));

void free_results(results_countpairs **results);
```

5.1.2 API for $\xi(r_p, \pi)$

```
typedef struct{
  uint64_t *npairs;
  DOUBLE *rupp;
  DOUBLE *rpavg;
  DOUBLE pimax;
  int nbin;
  int npibin;
} results_countpairs_rp_pi;

results_countpairs_rp_pi * countpairs_rp_pi(
const int64_t ND1, const DOUBLE *X1, const DOUBLE *Y1, const DOUBLE *Z1,
const int64_t ND2, const DOUBLE *X2, const DOUBLE *Y2, const DOUBLE *Z2,
#ifdef USE_OMP
const int numthreads,
#endif
const int autocorr,
const char *binfile,
const double pimax)  __attribute__((warn_unused_result));

void free_results_rp_pi(results_countpairs_rp_pi **results);
```

5.1.3 API for $w_p(r_p)$

```
typedef struct{
  uint64_t *npairs;
  DOUBLE *wp;
  DOUBLE *rupp;
  DOUBLE *rpavg;
  DOUBLE pimax;
  int nbin;
} results_countpairs_wp;
```

```
results_countpairs_wp *countpairs_wp(
const int64_t ND1, DOUBLE * restrict X1, DOUBLE * restrict Y1, DOUBLE *
    restrict Z1,
const double boxsize,
#ifdef USE_OMP
const int numthreads,
#endif
const char *binfile,
const double pimax) __attribute__((warn_unused_result));


void free_results_wp(results_countpairs_wp **results);
```

5.2 Python Bindings

The `python_bindings` directory contains python bindings for python 2.x. Note that python3 is not supported out of the box[6]. If all went well, then typing `python call_correlation_functions.py` should run the example python code. If you get an error (and you are on a MAC), then refer to Section 2.4 or the FAQ.
If you edit the `common.mk` file and compile for `double precision` arithmetic, then be sure to change the line:

```
dtype=np.float32
```

to

```
dtype=np.float64.
```

Otherwise, you will get a `TypeError` at runtime.

6 Extending the Code

6.1 Different Type of Input Data File

All of the codes use `io.c` in the `io` sub-directory to read-in the data. If you want to specify a different file format, the easiest way would be to edit `io.c`. Decide on the file format code and add another `strncmp` case in `io.c`. Remember that the `x/y/z` are declared as `void` pointers, so you can not directly reference the `x/y/z` pointers. If you do add support for a different file-type, please submit a pull request and I will be happy to merge it into the code-base.

[6] In the future, I might switch to cython to cover both python2 and python3

6.2 Computing a different type of correlation function

Let's say, you want to compute a marked correlation function. Now, you will need to read-in/create the marks for each individual point. And then you will have to add an appropriate field to the `cellarray` structure (see 4.1).

6.3 Using SSE instead of AVX

If your CPU is too old and does not support AVX, then you can still use SSE instrinsics to compute the correlation functions. However, this will require replacing all of the AVX sections with corresponding SSE intrinsics. Email me and I will guide you through the conversion process.