Users Guide for Corrfunc

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

January 28, 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.
Cache locality and hand-written AVX intrinsics are the reasons why the code is very fast. However, that also means that the code is not very portable. I have tried my best to ensure that the codes work on Linux and MAC OSX. If it does not work for you, particularly if you are on a reasonable Linux install, please email me. The API for the codes can be considered frozen; I will not change the API without changing the MAJOR release version.
The paper associated with the codes is being (slowly) written by me. I hope to submit the paper to Astronomy & Computing and release the codes simultaneously. Once I submit the paper to arXiv, there will be an actual paper arXiv link here.

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.g 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

[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 much harder to modify/maintain by any one other than me !

minor release versions (current `$MAJOR=1`, `$MINOR=0`). I will only change the `$MAJOR` version in the highly unlikely event that the API changes.

2.2 Directory Structure

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.
    ├── xi_of_r........................Source files for ξ(r).
    ├── xi_rp_pi.......................Source files for ξ(r_p, π).
    └── wp.............................Source files for w_p(r_p).
```

2.3 Compilation 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. I 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

Table I. *List of compilations options, what the options mean and their dependencies for the codes.*

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 $2\times$, $\xi(r_p, \pi)$ 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.

may have to do more to get the library to work – so I have outlined some of the scenarios in Section 2.5.

2.4 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.

2.5 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
  ```

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

- 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.

2.6 Running the tests

If installation went fine, then run `make tests` to run a suite of tests. If any of the tests fail, then please email me. I have never seen the tests fail unless I made some coding error while modifying the source. Once the tests pass successfully, type `make install` to install the binaries into the `bin` directory.

3 Running the Codes

The codes should run straight out of the box. Note, the $\xi(r)$ and $\xi(r_p, \pi)$ can compute both auto and cross-correlations with and without PERIODIC boundary conditions whereas the $w_p(r_p)$ code only computes the auto-correlation with PERIODIC boundary conditions. Also, since both the $\xi(r)$ and $\xi(r_p, \pi)$ codes compute cross-correlations, pairs are *double-counted* in both $\xi(r)$ and $\xi(r_p, \pi)$ codes. You could, in theory, make the auto-correlation bit faster by only computing unique pairs (as is done in $w_p(r_p)$, see Section 4.3.4 for details). However, since the total run-time will dominated by the cross-correlation (the number of randoms is typically an order of magnitude larger than the number of data points), I have not implemented those optimizations for the auto-correlation calculations.

3.1 Input File Formats

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:

Listing 3.1 The header format for `fast-food` files

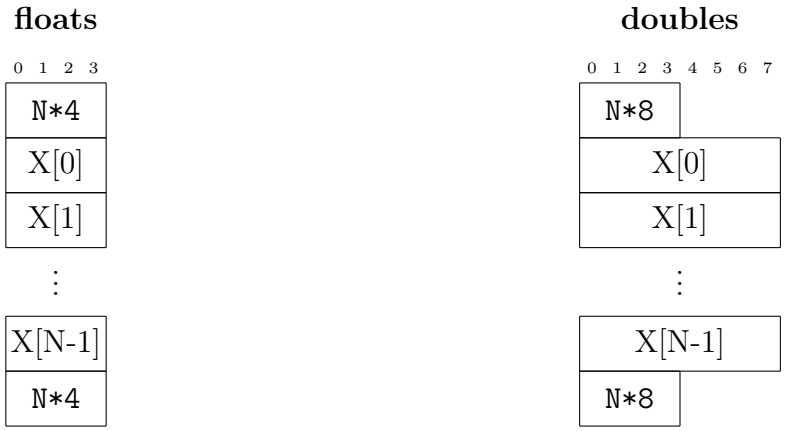
```
int idat[5];
float fdat[9];
float znow;
```

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.

0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15 16 17 18 19 20 21 22 23	24 25 26 27
20	Int	N	3 Integers	20

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.* The blocks for `Y/Z` follow after the `X` block.

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]
```

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 > 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. Note, that $\xi(r)$ double-counts the pairs (as does $\xi(r_p, \pi)$). The

[4] I assume bins are non-overlapping

inputs to DD are:

- file1 – the file name for the first file.
- format1 – the file format for the first file. Options are a,c,f – see Section 3.1.
- file2 – the file name for the second file.
- format2 – the file format for the second file. Options are a,c,f – see Section 3.1.
- 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 rows; each row contains the 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. Note, that $\xi(r_p, \pi)$ double-counts the pairs (as does $\xi(r)$).The inputs to DDrppi are:

- file1 – the file name for the first file.
- format1 – the file format for the first file. Options are a,c,f – see Section 3.1.
- file2 – the file name for the second file.
- format2 – the file format for the second file. Options are a,c,f – see Section 3.1.
- 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` rows; each row contains the 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. Options are `a,c,f` – see Section 3.1.
- `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` rows; each row contains the 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 $\pi_{\max}$.
- In each 3-D cell, store all the particles that are located in that cell in contiguous `X/Y/Z` arrays (see Listing 4.4). The particle data are contained in a structure defined in Listing 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 `j` loop in `countpairs.c, countpairs_rp_pi.c, countpairs_wp.c`.
- 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 Partitioning the Particles based on r_{max}

We need to compute pairwise distances to get the correlation function. A naive implementation of a correlation function would compute *all possible* pairwise separations with a complexity $\mathcal{O}(N^2)$. However, for almost all correlation functions, we are only interested in separations less than a certain r_{max}, where r_{max} is much smaller than the domain of the point distribution itself. We can then immediately see a way to prune pairs that can not *possibly*

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

be within r_{max}. If we impose a 3-d grid, with cell-size r_{max}, then two points separated by more than one cell size (r_{max}) in any one dimension can not be within r_{max} of each other (see Fig. 1 for a 2-D representation). Thus, given one point which is the target galaxy and a grid with cell-size r_{max}, immediately allows us to prune *all* of the points that are not within 1 cell offset in each dimension. However, even with this pruning, the actual implementation of the algorithm matters. For instance, the non-AVX version of the codes tend to run $2-3\times$ slower than the AVX version.

4.2 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.

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.

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

The code `gridlink.c` takes in an input list of 3 arrays `X/Y/Z` and grids them into a regular 3-D grid using the specified bins parameter `max_x_size` or `max_y_size` or `max_z_size` for the `X/Y/Z` axes respectively. [6] Once the number of grid cells along each axes has been determined, we allocate memory for the `struct lattice`. This `struct lattice` is a declared as an 1-D array; the conversion from the three indices in 3-D (`ix,iy,iz`) to a single index (`index`) happens through the last line in Listing 4.2. Such an 1-D array for `struct lattice` gives much better OpenMP scaling.

Listing 4.2 Accessing as `lattice[index]` rather than `lattice[ix][iy][iz]`.

```
int64_t totncells  = nmesh_x * nmesh_y * nmesh_z;
cellarray *lattice  = my_malloc(sizeof(cellarray), totncells);
int64_t *nallocated = my_malloc(sizeof(*nallocated), totncells);
int64_t index = ix*nmesh_y*nmesh_z + iy*nmesh_z + iz;
```

[6] In practice, the bins may be further subdivided using the corresponding bin refine factors. These bin refine factors seem to influence runtime the most, you should experiment with a few values of `bin_refine_factor` and `zbin_refine_factor` to see what produces the best runtimes for your typical scenario.

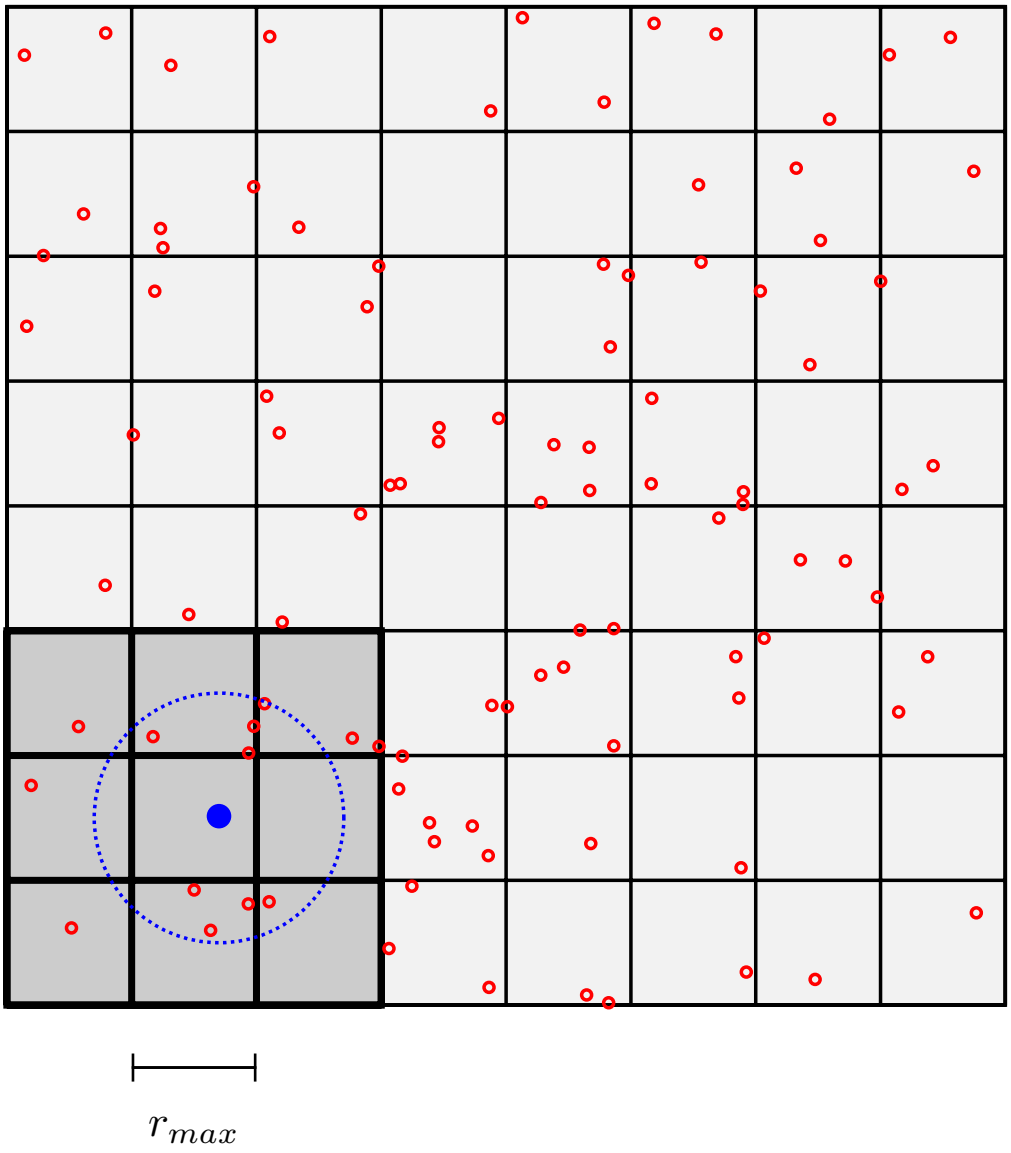


Figure 1. *A 2-D grid showing the bin-lattice partitioning scheme. The bigger square show the entire domain, the red circles show a random distribution of 100 particles. Let's say we want to compute all pairs for the target blue point, then we would only have to consider red points that are within one 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.*

Now that we know the total number of cells in the entire domain, we need to allocated memory to store the particles in each cell. However, there is no way to know the exact number of particles in each cell without processing the entire data-set; so, we pre-allocate with an estimate of the expected number of particles from the volume of each 3-D cell (assuming a random distribution). Then, we can allocate memory for each of the `X/Y/Z` arrays inside each 3-D cell:

Listing 4.3 Pre-allocating memory for the `X/Y/Z` arrays in `struct cellarray`.

```
for (int64_t index=0;index<totncells;index++) {
  lattice[index].x = my_malloc(sizeof(DOUBLE),expected_n);
  lattice[index].y = my_malloc(sizeof(DOUBLE),expected_n);
  lattice[index].z = my_malloc(sizeof(DOUBLE),expected_n);
  lattice[index].nelements=0;
  nallocated[index] = expected_n;
}
```

Here, `nallocated` is an array that keeps track of the amount of memory alread allocated for each cell. After this step, all cells have been allocated memory for `expected_n` particles. Now, we can begin to process the individual particles and assigning them to the 3-D cells (if enough memory has already been allocated to assign the new particle).

Listing 4.4 Assigning the particles to the `struct cellarray` in the cell.

```
for (int64_t i=0;i<np;i++)  {
  ix=(int)((x[i]-xmin)*xinv) ;
  iy=(int)((y[i]-ymin)*yinv) ;
  iz=(int)((z[i]-zmin)*zinv) ;

  int64_t index = ix*nmesh_y*nmesh_z + iy*nmesh_z + iz;

  if(lattice[index].nelements == nallocated[index]) {
    expected_n = nallocated[index]*MEMORY_INCREASE_FAC;

    lattice[index].x = my_realloc(lattice[index].x
       ,sizeof(DOUBLE),expected_n,''lattice.x'');
    lattice[index].y = my_realloc(lattice[index].y
       ,sizeof(DOUBLE),expected_n,''lattice.y'');
    lattice[index].z = my_realloc(lattice[index].z
       ,sizeof(DOUBLE),expected_n,''lattice.z'');

    nallocated[index] = expected_n;
  }
  int64_t ipos=lattice[index].nelements;
  lattice[index].x[ipos] = x[i];
```

```
    lattice[index].y[ipos] = y[i];
    lattice[index].z[ipos] = z[i];
    lattice[index].nelements++;
  }
```

The loop goes over all of the particles and calculates the corresponding 3-D cell indices — `ix,iy,iz`. With these 3 variables, the corresponding 1-D index, `index`, can be calculated. The next lines check (and reallocate memory, if necessary) to ensure that enough memory has been allocated to the `struct lattice[index]` to accommodate this new particle. The last 5 lines are simply assigning the particle into the appropriate `struct lattice[index]`. Once all the particles have been assigned, the cells contain contiguous `X/Y/Z` arrays describing the original particle distribution.

4.3 The Pair-Counting Algorithms

After running through `gridlink`, the particle distribution is stored in contiguous `X/Y/Z` inside the 1-D array of `struct cellarray`. To find all possible pairs, we first need to loop over all particles in the first data-set.

Listing 4.5 Looping over all cells in the first data-set.

```
for (int64_t index1=0;index1<totncells;index1++) {
  const cellarray *first = &lattice1[index1];
  const DOUBLE *x1 = first->x;
  const DOUBLE *y1 = first->y;
  const DOUBLE *z1 = first->z;
}
```

Now, we have the target cell pointer and the associated `x1/y1/z1` array pointers. Next, we need to get the indices for the neighbouring cells in 3-D. In order to do that, first the 1-D index, `index1` needs to be converted into a set of three 3-D indices, `ix, iy, iz`. Listing 4.6 shows the conversion from the 1-D index to the corresponding 3-D indices.

Listing 4.6 Reconstructing 3-D index for `first` cell in the first data-set.

```
const int iz = index1 % nmesh_z ;
const int ix = index1 / (nmesh_z * nmesh_y) ;
const int iy = (index1 - iz - ix*nmesh_z*nmesh_y)/nmesh_z ;
```

After executing the code segment in Listing 4.6, we have the full 3-D indices for the target cell. Now, we have to find all of the indices for the neighbouring cells that can potentially satisfy the r_{max} constraint. This requires considering all 3-D cells that are located within `bin_refine_factor` of the target cell (for each dimension). Since the target cell, `first` has a 3-D X index of `ix`, this means all cells that have X indices in the range `ix` $\pm$

`bin_refine_factor` can potentially have pairs that satisfy the r_{max} constraint. However, in case of PERIODIC boundary conditions, we also have to ensure that the indices (and the actual particle positions) wrap around on the other side of the cube. Listing 4.7 shows how the looping over neighbouring cells is done for the X dimension. Similar segments follow in the actual code for the Y/Z dimensions.

Listing 4.7 Looping over all the neighbouring cells and taking care of PERIODIC boundary conditions.

```
for(int iix=-bin_refine_factor;iix<=bin_refine_factor;iix++){
  int iiix;
#ifdef PERIODIC
  DOUBLE off_xwrap=0.0;
  if(ix + iix >= nmesh_x) {
    off_xwrap = -xdiff;
  } else if (ix + iix < 0) {
    off_xwrap = xdiff;
  }
  iiix=(ix+iix+nmesh_x)%nmesh_x;
#else
  iiix = iix+ix;
  if(iiix < 0 || iiix >= nmesh_x) {
    continue;
  }
#endif
  ⋮
  Similar chunks of code for Y/Z.
  ⋮
  const int64_t index2 = iiix*nmesh_y*nmesh_z + iiiy*nmesh_z +
     iiiz;
```

Once all of the three 3-D indices for the neighbouring cell has been determined, we can reconstruct the 1-D index, `index2` for that cell. With this 1-D index, we can create a pointer, `second`, that contains the `cellarray pointer` to the neighbouring cell. In Listing 4.8, we show how the neighbouring cell and the associated `x2/y2/z2` array pointers are defined.

Listing 4.8 Dereferencing the pointers for the neighbouring (`second`) cell under consideration.

```
const cellarray *second = &lattice2[index2];
const DOUBLE *x2 = second->x;
const DOUBLE *y2 = second->y;
const DOUBLE *z2 = second->z;
```

At this point, we have a set of `x1/y/1/z1` arrays with `first->nelements` elements representing the first data-set. We also have another set of `x2/y2/z2` arrays with `second->nelements` elements representing the second data-set. Now, we have to compute all possible pair-wise separations between these two data-sets. We begin with a loop over the elements in `first`:

Listing 4.9 Looping over all particles in the `first` cell and accounting for PERIODIC boundary conditions.

```
for(int64_t i=0;i<first->nelements;i++) {
  DOUBLE x1pos=x1[i];
  DOUBLE y1pos=y1[i];
  DOUBLE z1pos=z1[i];
#ifdef PERIODIC
  x1pos += off_xwrap;
  y1pos += off_ywrap;
  z1pos += off_zwrap;
#endif
```

If PERIODIC boundary conditions are enabled, then `off_xwrap`, `off_ywrap` and `off_zwrap` have been declared and initialized in Listing 4.7. By wrapping the elements if the first data-set, we can avoid the wrapping operations in the `j-loop` over all particles in `second`.

Listing 4.10 AVX intrinsics for looping over all particles in the `second` cell. PERIODIC boundary conditions have already been accounted for in `x1pos,y1pos,z1pos` variables.

```
const AVX_FLOATS m_x1pos = AVX_SET_FLOAT(x1pos);
const AVX_FLOATS m_y1pos = AVX_SET_FLOAT(y1pos);
const AVX_FLOATS m_z1pos = AVX_SET_FLOAT(z1pos);

int64_t j;
for(j=0;j<=(second->nelements-NVEC);j+=NVEC) {
  const AVX_FLOATS x2pos = AVX_LOAD_FLOATS_UNALIGNED(&x2[j]);
  const AVX_FLOATS y2pos = AVX_LOAD_FLOATS_UNALIGNED(&y2[j]);
  const AVX_FLOATS z2pos = AVX_LOAD_FLOATS_UNALIGNED(&z2[j]);

  const AVX_FLOATS m_xdiff = AVX_SUBTRACT_FLOATS(m_x1pos,x2pos);
  const AVX_FLOATS m_ydiff = AVX_SUBTRACT_FLOATS(m_y1pos,y2pos);
  const AVX_FLOATS m_zdiff = AVX_SUBTRACT_FLOATS(m_z1pos,z2pos);
```

The three codes $\xi(r)$, $\xi(r_p,\pi)$ and $w_p(r_p)$ diverge somewhat after this point. For $\xi(r)$, we need to calculate the full 3-D separation, whereas for $\xi(r_p,\pi)$ and $w_p(r_p)$ the separation is the projected distance. I will discuss further implementations in the following sub-sections dedicated to each code.

4.3.1 Pair-counting for $\xi(r)$

Pair-counting for $\xi(r)$ is the most straight-forward. The squared 3-D separation, `r2`, is simply the sum of the squared differences in each `X/Y/Z` dimensions.

Listing 4.11 Calculating squared separations in $\xi(r)$.

```
const AVX_FLOATS m_xdiff_sqr = AVX_SQUARE_FLOAT(m_xdiff);
const AVX_FLOATS m_ydiff_sqr = AVX_SQUARE_FLOAT(m_ydiff);
const AVX_FLOATS m_zdiff_sqr = AVX_SQUARE_FLOAT(m_zdiff);
const AVX_FLOATS m_xydiff_sqr_sum =
   AVX_ADD_FLOATS(m_xdiff_sqr,m_ydiff_sqr);
AVX_FLOATS r2 = AVX_ADD_FLOATS(m_zdiff_sqr,m_xydiff_sqr_sum);
```

Once all the `NVEC` separations have been computed, we use bit-masks to check if any separations fall within the range `sqr_rpmin` and `sqr_rpmax`. If not, we `continue` with the `j-loop`.

Listing 4.12 Bit-masks in $\xi(r)$.

```
m_mask_left = AVX_COMPARE_FLOATS(r2,m_sqr_rpmax,_CMP_LT_OS);
if(AVX_TEST_COMPARISON(m_mask_left) == 0) {
  continue;
}

const AVX_FLOATS m_mask = AVX_BITWISE_AND(m_mask_left,
   AVX_COMPARE_FLOATS(r2, m_sqr_rpmin, _CMP_GE_OS));
if(AVX_TEST_COMPARISON(m_mask) == 0) {
  continue;
}
r2 = AVX_BLEND_FLOATS_WITH_MASK(m_sqr_rpmax, r2, m_mask);
m_mask_left = AVX_COMPARE_FLOATS(r2,m_sqr_rpmax,_CMP_LT_OS);
```

If the code reaches past this bit-mask section, then at least one separation is within range. In that case, we use the code in Listing 4.18 to update the `npairs` array (and the `rpavg` array if OUTPUT_RPAVG is enabled).

4.3.2 Pair-separations for the projected correlation functions $\xi(r_p,\pi)$,$w_p(r_p)$

Listing 4.13 shows the squared distance calculation for $\xi(r_p,\pi)$ and $w_p(r_p)$. `r2` is simply the projected separation in the `X-Y` plane.

Listing 4.13 Calculating squared separations in $\xi(r_p,\pi)$ and $w_p(r_p)$.

```
const AVX_FLOATS m_xdiff_sqr = AVX_SQUARE_FLOAT(m_xdiff);
const AVX_FLOATS m_ydiff_sqr = AVX_SQUARE_FLOAT(m_ydiff);
```

```
AVX_FLOATS r2 = AVX_ADD_FLOATS(m_xdiff_sqr,m_ydiff_sqr);
```

4.3.3 Pair-counting for $\xi(r_p, \pi)$

Listing 4.14 Bit-masks in $\xi(r_p, \pi)$.

```
const AVX_FLOATS m_mask_pimax =
   AVX_COMPARE_FLOATS(m_zdiff,m_pimax,_CMP_LT_OS);
const int test = AVX_TEST_COMPARISON(m_mask_pimax);
if(test == 0) {
  continue;
}

const AVX_FLOATS m1 =
   AVX_COMPARE_FLOATS(r2,m_sqr_rpmin,_CMP_GE_OS);
r2 = AVX_BLEND_FLOATS_WITH_MASK(m_sqr_rpmax,r2,m_mask_pimax);

m_mask_left = AVX_COMPARE_FLOATS(r2,m_sqr_rpmax,_CMP_LT_OS);
const AVX_FLOATS m_mask = AVX_BITWISE_AND(m1,m_mask_left);
int test1 = AVX_TEST_COMPARISON(m_mask);
if(test1 == 0) {
  continue;
}
m_zdiff = AVX_BLEND_FLOATS_WITH_MASK(m_pimax, m_zdiff, m_mask);
#ifdef OUTPUT_RPAVG
union_mDperp.m_Dperp = AVX_SQRT_FLOAT(r2);
#endif
union_pibin.m_ibin =
   AVX_TRUNCATE_FLOAT_TO_INT(AVX_MULTIPLY_FLOATS(m_zdiff,m_inv_dpi));
```

In the $\xi(r_p, \pi)$ code, we have to update a two dimensional `npairs(`r_p`,`π`)` array. This means we can not directly update the `npairs` matrix and instead have to use a separate loop (this loop is typically only invoked for $\xi(r)$ and $w_p(r_p)$ when OUTPUT_RPAVG is enabled). As a result of this extra loop, $\xi(r_p, \pi)$ is $2 - 3\times$ slower than $\xi(r)$ and $w_p(r_p)$.

Listing 4.15 Updating the npairs matrix in $\xi(r_p, \pi)$.

```
for(int jj=0;jj<NVEC;jj++) {
  int rpbin = union_rpbin.ibin[jj];
  int pibin = union_pibin.ibin[jj];
  int ibin = rpbin*(npibin+1) + pibin;
  npairs[ibin]++;
#ifdef OUTPUT_RPAVG
```

```
    rpavg [ibin] += union_mDperp.Dperp[jj];
#endif
}
```

4.3.4 Pair-counting for $w_p(r_p)$

Since the pair-counting in $w_p(r_p)$ always assumes PERIODIC boundary conditions *and* only computes an auto-correlation, extra optimizations are possible in the $w_p(r_p)$ calculation. First, each individual cell in the `struct lattice` elements are sorted based on their `Z` arrays. The `X/Y` arrays are also re-ordered simultaneously. Once all the `Z` values inside a cell have been sorted, we can avoid double-counting the pairs by changing the `iiz loop` to be:

Listing 4.16 Optimizing the loop over neighbouring cells `z` in $w_p(r_p)$.

```
for(int iiz=0;iiz<=zbin_refine_factor;iiz++)
```

instead of

```
for(int iiz=-zbin_refine_factor;iiz<=zbin_refine_factor;iiz++)
```

Another advantage of sorting the `Z` values is earlier termination inside the actual calculation. The Listing 4.17 shows how to break early from the `j-loop`. Since the `z2` values are always stored in increasing order, if *all* values of `zdiff:=z2[j:j+NVEC-1]-z1` are greater than $\pi_{\rm max}$, then none of the `zdiff` values in future iterations of the `j-loop` can be smaller than $\pi_{\rm max}$. When the code encounters such a scenario, it updates the `j` variable to `second->nelements` to ensure that the serial section of the code is not executed and breaks out the AVX `j-loop`.

Listing 4.17 AVX intrinsics for calculating separations in $w_p(r_p)$ and checking for early termination.

```
const AVX_FLOATS m_zdiff =
   AVX_SUBTRACT_FLOATS(m_z2,m_zpos);//z2[j:j+NVEC-1] - z1
AVX_FLOATS m_mask_pimax =
   AVX_COMPARE_FLOATS(m_zdiff,m_pimax,_CMP_LT_OS);
const int test = AVX_TEST_COMPARISON(m_mask_pimax);
if(test == 0) {
  j = second->nelements;
  break;
}
```

4.4 AVX intrinsics to update the npairs histogram

This section explains the AVX intrinsics used to update the pair-counts histogram, `npairs`.[7] If the code execution reaches this loop, at least one (squared) pair separation falls within the range `sqr_rpmin` and `sqr_rpmax`, where `sqr_rpmin` is the squared lower radial limit of the first bin and `sqr_rpmax` is the squared upper limit of the last bin (equivalent to $r_{max}{}^2$ in this user-guide). Since the pairs are more likely to occur in the largest separation bins, the loop goes backwards from the last bin to the first and uses early loop-termination in case all possible pairs have already been accounted for.

Listing 4.18 AVX intrinsics for updating the npairs histgram for $\xi(r)$ and $w_p(r_p)$.

```
for(int kbin=nrpbin-1;kbin>=1;kbin--) {
  const AVX_FLOATS m1 =
     AVX_COMPARE_FLOATS(r2,m_rupp_sqr[kbin-1],_CMP_GE_OS);
  const AVX_FLOATS m_bin_mask = AVX_BITWISE_AND(m1,m_mask_left);
  m_mask_left =
     AVX_COMPARE_FLOATS(r2,m_rupp_sqr[kbin-1],_CMP_LT_OS);
  const int test2  = AVX_TEST_COMPARISON(m_bin_mask);
  npairs[kbin] += AVX_BIT_COUNT_INT(test2);
#ifdef OUTPUT_RPAVG
  m_rpbin = AVX_BLEND_FLOATS_WITH_MASK(m_rpbin,m_kbin[kbin],
     m_bin_mask);
#endif
  const int test3 = AVX_TEST_COMPARISON(m_mask_left);
  if(test3 == 0) break;
}
```

Note that `rupp_sqr` (and its AVX equivalent, `m_rupp_sqr`) contains the squared upper limits for the bins. Thus, when considering bin `kbin`, `m_rupp_sqr[kbin-1]` gives the squared lower radial limit for the bin while `m_rupp_sqr[kbin]` gives the squared upper limit for bin `kbin`. Here, `m1` is the mask that contains all separations that satisfy `r2` $\geq$ `rupp_sqr[kbin-1]`, while the mask `m_mask_left` contains those squared separations that satisfy `r2 < rupp_sqr[kbin]`. Note, that `m_mask_left` is either computed before entering the `kbin` loop or during a previous iteration of the same `kbin` loop. The AVX variable `m_bin_mask` then contains the bitwise and of `m1` and `m_mask_left` – the mask for the squared separations that fall into `kbin`. The variable `test2` contains an integer composed of the upper set-bits of the mask `m_bin_mask` – thus, contains only 4/8 useful bits for `float/double` precision calculations respectively. The `npairs` pair-counts is then updated using a hardware `popcnt` instruction. The last two lines check if there are any more pairs left that satisfy the lower bin-ranges; if not, the loop is terminated with a `break` statement.

[7] The thread-local version in $w_p(r_p)$ is called `local_npair`.

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. Be absolutely sure to pass arrays for the correct type – float arrays if you did not use DOUBLE_PREC (default) or double arrays if you did use DOUBLE_PREC. Also, only include headers from the `xi_theory/include` directory – those headers correspond to the actual static library. *DO NOT include* the header files from the `xi_of_r/xi_rp_pi/wp` directories – these headers do not contain the correct function signature for the compilation flags used to generate the static libraries.

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 arrays (*of the correct* ***`float/double`*** *type*) into the functions. Make sure to include the static library in the linking step to create a stand-alone executable (see the `Makefile` in the `examples` directory).

5.1.1 API for $\xi(r)$

The interface for the 3-D correlation function, $\xi(r)$, is through the `countpairs` function. Here is the corresponding function signature:

Listing 5.1 API for the 3-D $\xi(r)$.

```
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);
```

The parameters to the function are:

- `ND1` – number of elements in the first data-set.
- `X1` – the array of X-values in the first data-set.
- `Y1` – the array of Y-values in the first data-set.

- `Z1` – the array of Z-values in the first data-set.
- `ND2` – number of elements in the second data-set.
- `X2` – the array of X-values in the second data set.
- `Y2` – the array of Y-values in the second data set.
- `Z2` – the array of Z-values in the second data set.
- `numthreads` – the number of threads to use (if USE_OMP is enabled in `common.mk`).
- `autocorr` – if an auto-correlation is being calculated (1 implies auto-correlation, flag is used for some runtime optimizations).
- `binfile` – file name that contains the radial bins. See Section 3.2 for details on how to create this file.

The output from `countpairs` is contained in `struct results_countpairs`. The structure definition is:

Listing 5.2 Structure definition for the output of $\xi(r)$.

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

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

The fields in the structure correspond to:

- `npairs` – array containing the pair-counts.
- `rupp` – array containing the upper limits of the bins. `rupp[0]` gives the lower-limit of the first radial bin.
- `rpavg` – array containing the average value of the separations for all the pairs that fell into the bin. Will contain meaningful values if OUTPUT_RPAVG is defined in `common.mk`; identically `0.0` otherwise.
- `nbin` – the number of radial bins used. Note that the actual pair-counts are stored in the index range `[1,nbin-1]`. The zero'th bin contains garbage for all of the arrays in this `struct results_countpairs`.

After the results structure has been used, use `free_results(&results_countpairs)` to free allocated memory.

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

The interface for the 2-D correlation function, $\xi(r_p, \pi)$, is through the `countpairs_rp_pi` function. Here is the corresponding function signature:

Listing 5.3 API for the 2-D $\xi(r_p, \pi)$

```
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);
```

The parameters to the function are:

- `ND1` – number of elements in the first data-set.
- `X1` – the array of X-values in the first data-set.
- `Y1` – the array of Y-values in the first data-set.
- `Z1` – the array of Z-values in the first data-set.
- `ND2` – number of elements in the second data-set.
- `X2` – the array of X-values in the second data set.
- `Y2` – the array of Y-values in the second data set.
- `Z2` – the array of Z-values in the second data set.
- `numthreads` – the number of threads to use (if USE_OMP is enabled in `common.mk`).
- `autocorr` – if an auto-correlation is being calculated (1 implies auto-correlation, flag is used for some runtime optimizations).
- `binfile` – file name that contains the radial bins. See Section 3.2 for details on how to create this file.
- `pimax` – the maximum line-of-sight distance (assumed to be the `Z` axis) to use.

The output from `countpairs_rp_pi` is contained in `struct results_countpairs_rp_pi`. The structure definition is:

Listing 5.4 Structure definition for the output of $\xi(r_p, \pi)$

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

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

The fields in the structure correspond to:

- `npairs` – array containing the pair-counts. Number of elements in the array is `(npibin+1)`$\times$`(nbin+1)`
- `rupp` – array containing the upper limits of the bins. `rupp[0]` gives the lower-limit of the first radial bin. Number of elements in the array is `nbin`.
- `rpavg` – array containing the average value of the separations for all the pairs that fell into the bin. Will contain meaningful values if OUTPUT_RPAVG is defined in `common.mk`; identically `0.0` otherwise. Number of elements in the array is `(npibin+1)`$\times$`(nbin+1)`.
- `pimax` – the maximum line-of-sight distance used in the bins.
- `nbin` – the number of radial bins used. Note that the actual pair-counts are stored in the index range `[1,nbin-1]`. The zero'th bin contains garbage for all of the arrays in this `struct results_countpairs_rp_pi`.
- `npibin` – the number of π bins used. The total number of elements in the arrays in the `struct countpair_rp_pi` is `(npibin+1)`$\times$`(nbin+1)`. As usual, meaningful data is only contained in the radial bin range `[1,nbin-1]`.

After the results structure has been used, use `free_results_rp_pi(&results_countpairs_rp_pi)` to free allocated memory.

5.1.3 API for $w_p(r_p)$

The interface for the projected correlation function, $w_p(r_p)$, is through the `countpairs_wp` function. The code *always* uses PERIODIC boundary conditions, irrespective of the settings in `common.mk`. Here is the corresponding function signature[8]:

[8]The `X1/Y1/Z1` arrays are not declared with `const` qualifiers because I sort those arrays on `Z` in the `countpairs_wp` function.

Listing 5.5 API for the $w_p(r_p)$.

```
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);
```

The parameters to the function are:

- `ND1` – number of elements in the data-set.
- `X1` – the array of X-values in the data-set.
- `Y1` – the array of Y-values in the data-set.
- `Z1` – the array of Z-values in the data-set.
- `boxsize` – the boxsize that fully contains the `X/Y/Z` values.
- `numthreads` – the number of threads to use (if USE_OMP is enabled in `common.mk`).
- `binfile` – file name that contains the radial bins. See Section 3.2 for details on how to create this file.
- `pimax` – the maximum line-of-sight distance (assumed to be the `Z` axis) to use.

The output from `countpairs_wp` is contained in `struct results_countpairs_wp`. The structure definition is:

Listing 5.6 Structure definition for the output of $w_p(r_p)$

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

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

The fields in the structure correspond to:

- `npairs` – array containing the pair-counts. Number of elements in the array is `nbin`.
- `wp` – array containing the actual `wp` values.
- `rupp` – array containing the upper limits of the bins. `rupp[0]` gives the lower-limit of the first radial bin. Number of elements in the array is `nbin`.
- `rpavg` – array containing the average value of the separations for all the pairs that fell into the bin. Will contain meaningful values if OUTPUT_RPAVG is defined in `common.mk`; identically `0.0` otherwise. Number of elements in the array is `nbin`.
- `pimax` – the maximum line-of-sight distance used in the bins.
- `nbin` – the number of radial bins used. Note that the actual pair-counts are stored in the index range `[1,nbin-1]`. The zero'th bin contains garbage for all of the arrays in this `struct results_countpairs_wp`.

After the results structure has been used, use `free_results_wp(&results_countpairs_wp)` to free allocated memory.

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[9]. 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 2.5 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 Benchmarks & Scaling

In this section we present the runtimes and scalings for different number of particles, r_{max} and OpenMP threads for the codes. For all of the scaling tests, only an auto-correlation calculation was used and the fiducial catalog contains $\sim$ 1.2 million galaxies on a periodic cube of side 420 $h^{-1}Mpc$.

6.1 Scaling with Number of Particles

In Fig. 2, we show the scaling for the three codes with the number of particles. For this scaling, we subsampled the fiducial mock to attain 10 logarithmic steps in particle number ranging from 1.2×10^4 to 1.2×10^6. All the timings are generateed using 1 thread.

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

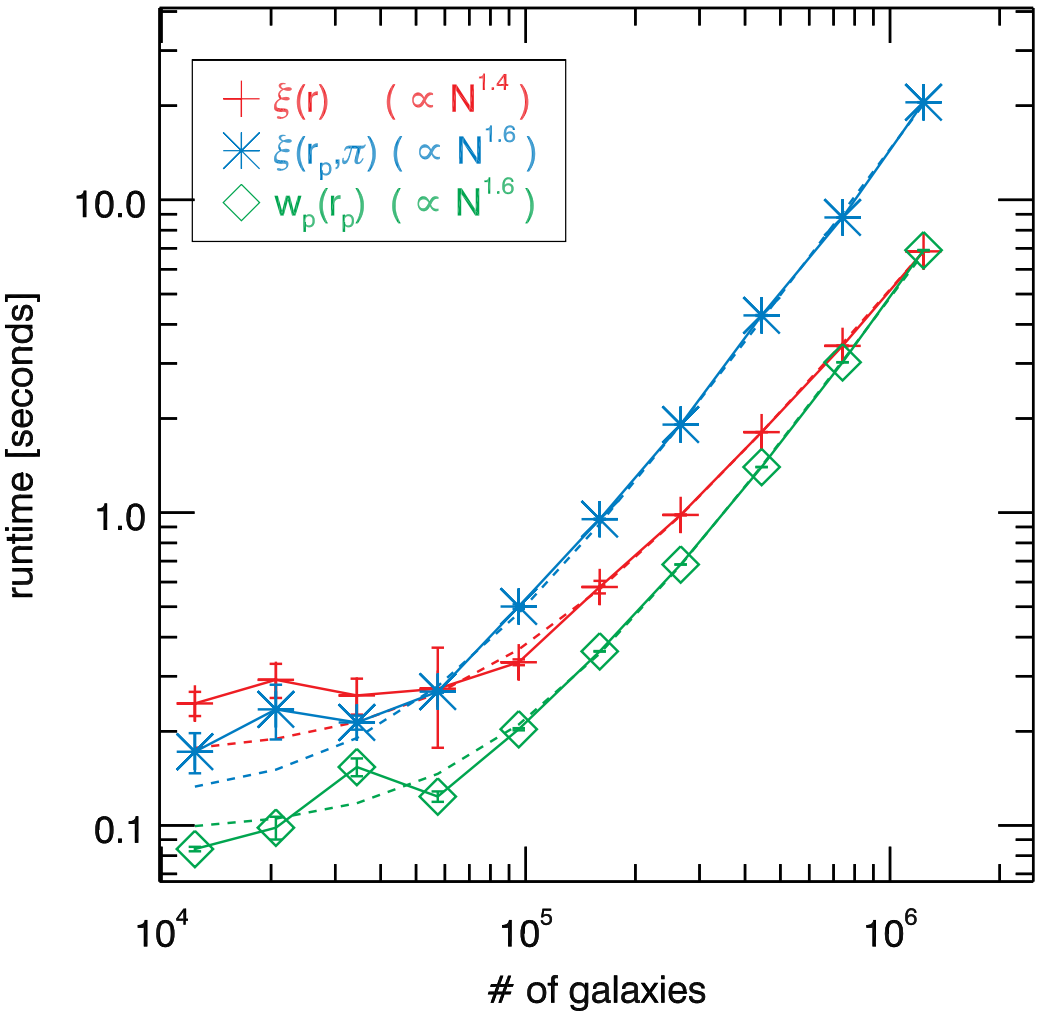


Figure 2. *Scaling with particle number for $\xi(r)$, $\xi(r_p, \pi)$, $w_p(r_p)$. The timings are obtained using 1 OpenMP thread.*

Table II. *OpenMP scaling for the three codes. Efficiencies are defined as* `Speedup/Nthreads`*. Note, that* $\xi(r)$ *has super-linear scaling with* `nthreads`*.*

Nthreads	Efficiency[%]		
	$\boldsymbol{\xi(r)}$	$\boldsymbol{\xi(r_p,\pi)}$	$\boldsymbol{w_p(r_p)}$
1	100	100	100
2	108	98	97
3	109	96	95
4	105	95	91
5	104	94	93
6	103	94	86
7	102	90	88
8	99	87	84
9	96	87	80
10	96	86	83
11	95	84	80
12	93	82	83
13	91	81	75
14	90	80	70
15	89	77	70
16	71	76	65

6.2 Scaling with r_{max}

The code runtime increases drastically as the largest requested separation, r_{max} increases. Roughly speaking, the runtimes scale as $\mathcal{O}(r_{max}^2)$ with $\xi(r)$ showing the strongest dependence on r_{max}. This is from the $\pi_{\rm max}$ dependence of $\xi(r_p,\pi)$ and $w_p(r_p)$. Even when r_{max} is large, the effective number of particles per cell only grows as r_{max}^2 and not r_{max}^3 (as it does for $\xi(r)$). If we extend the benchmarks to larger r_{max}, we will see a $\mathcal{O}(r_{max}^3)$ dependence for

6.3 Scaling with OpenMP threads

All of the codes presented here scale reasonably well (efficiency $\gtrsim 80\%$) up to 10 OpenMP threads. Beyond that, the work-load is scaling efficiency starts dropping off and plateaus for `nthreads` $\gtrsim 20$.

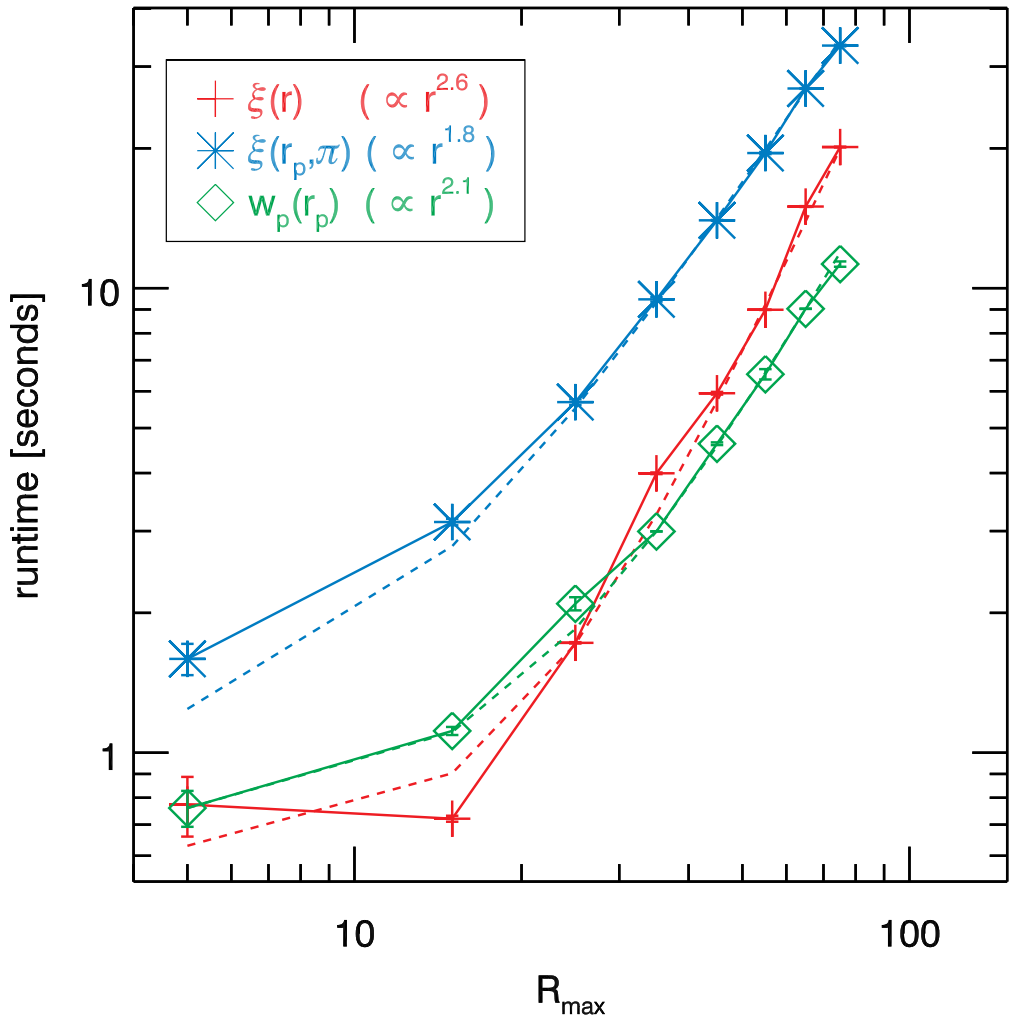


Figure 3. *Scaling with r_{max} for $\xi(r)$, $\xi(r_p, \pi)$, $w_p(r_p)$. The timings are obtained with 4 OpenMP threads.*

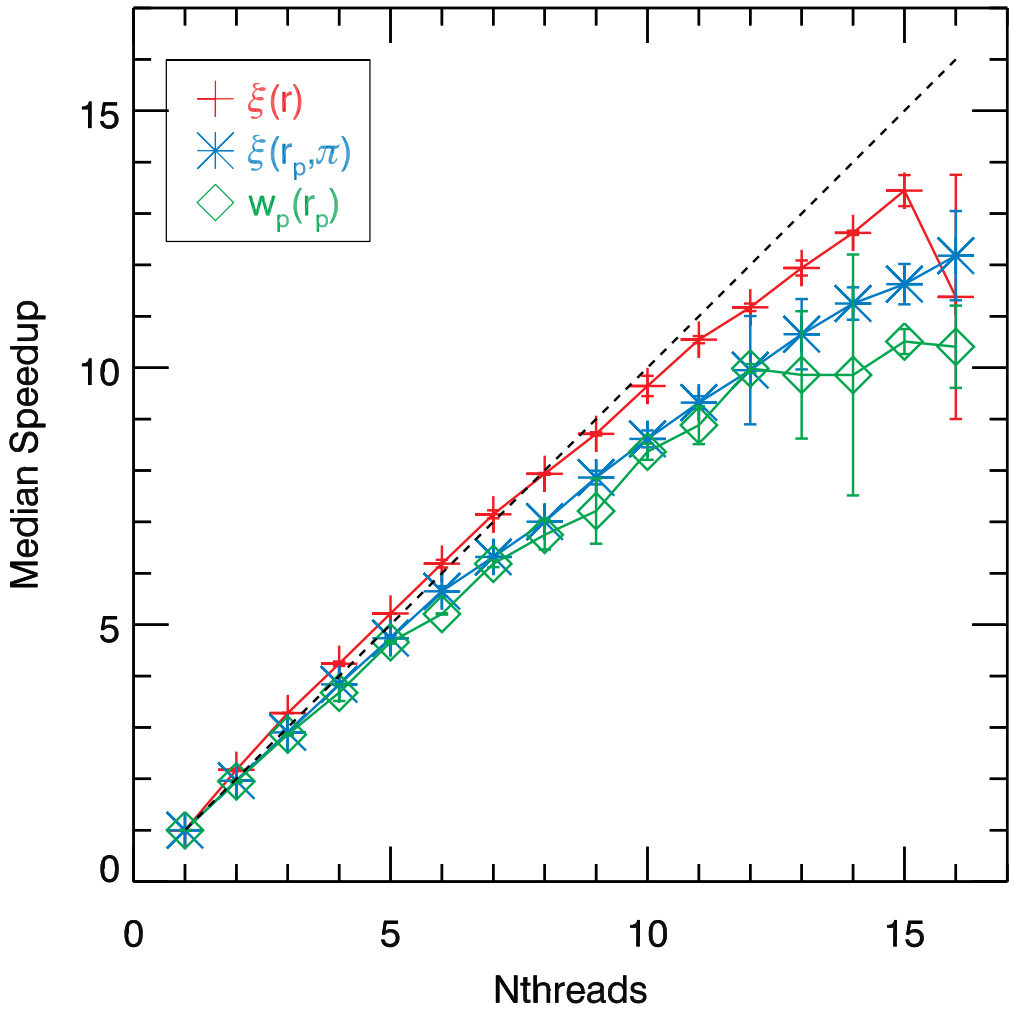


Figure 4. *OpenMP scaling for $\xi(r)$, $\xi(r_p, \pi)$, $w_p(r_p)$. The fiducial mock used here contains $\sim 10^6$ particles in a 420.0 $h^{-1}Mpc$ cube.*

7 Extending the Code

7.1 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.

7.2 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. Here are the steps you will need to create your custom correlation function code:

1. Add the data fields into the cellarray structure definition (see Listing 4.1.
2. Add in the memory allocation for the fields in `gridlink.c` after the `malloc` for `X/Y/Z` pointers (see Listing 4.3).
3. Extend `gridlink.c` to accept additional arrays and assign those arrays into `struct lattice` (see Listing 4.4).
4. Enable OUTPUT_RPAVG and DOUBLE_PREC (this is not required, but probably the easiest way to create a custom correlation function calculation).
5. Declare and zero-initialize a `results` array that will contain your custom correlation function. Follow the implementation in the code for calculating `rpavg`.
6. Add in the AVX arrays that load your custom data fields in the `j loop` in the `countpairs*` functions.
7. Add your custom correlation function weight into the `results` array. Combine thread-local arrays into a global one in case USE_OMP is selected.
8. Add the custom correlation function field to the corresponding `struct results_countpairs*`. Assign the `results` array into the `struct results_countpairs*`.
9. Add a call to `free(your field)` in the corresponding function `free_results_countpairs*`.

7.3 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.

8 License

The code has been released under the MIT License.