Corrfunc Documentation

Release 2.1.0

Manodeep Sinha <manodeep@gmail.com>

Aug 17, 2018

Contents

Corrfunc is a set of high-performance routines to measure clustering statistics. The main features of Corrfunc are:

- **Fast** All theory pair-counting is at least an order of magnitude faster than all existing public codes. Particularly suited for MCMC.
- **OpenMP Parallel** All pair-counting codes can be done in parallel (with strong scaling efficiency >~ 95% up to 10 cores)
- **Python Extensions** Python extensions allow you to do the compute-heavy bits using C while retaining all of the user-friendliness of python.
- **Modular** The code is written in a modular fashion and is easily extensible to compute arbitrary clustering statistics.
- **Future-proof** As I get access to newer instruction-sets, the codes will get updated to use the latest and greatest CPU features.

The source code is publicly available at https://github.com/manodeep/Corrfunc.

CHAPTER 1

Overview of Corrfunc

1.1 Package Installation

To install Corrfunc, you can either use pip or clone the repo from GitHub and build the source code. Either way, be sure to read the *Dependencies* section prior to installation.

1.1.1 Using pip

The simplest way to install the latest release of the code is with pip. Before installation, be sure you have installed the package dependencies described in the *Dependencies* section

```
pip install Corrfunc
```

This will install the latest official release of the code. If you want the latest master branch, you will need to build the code from source following the instructions in the next section.

1.1.2 Building from source

If you don't install the latest release using pip, you can instead clone the cource code and call the setup file. Before installation, be sure you have installed the package dependencies described in the *Dependencies* section. The first step is to clone the Corrfunc repository

```
git clone https://github.com/manodeep/Corrfunc.git
cd Corrfunc
make install
python setup.py install
```

1.1.3 Dependencies

The command-line version of Corrfunc needs the following packages to be installed:

- make: 3.80 or later
- C compiler: gcc >=4.6, clang, icc. Multi-threading will be disabled if the compiler does not support OpenMP.
- gsl: any recent version

If you plan to use the C extensions, then the following are required:

- Python: 2.6 or later
- Numpy: 1.7 or later

Any of the above can be installed with either pip or conda.

1.1.4 Verifying your installation

After installing Corrfunc, you should run the integrated test suite to make sure that the package was installed correctly. If you installed from source, then type the following in the root package directory,

```
make tests
```

If you installed using pip/conda, then use the following to run the tests

```
from Corrfunc.tests import tests
tests()
```

Once you have installed the package, see *Getting started with Corrfunc* for instructions on how to get up and running.

1.2 Getting started with Corrfunc

Corrfunc is a set of high-performance routines to measure clustering statistics. The codes are divided conceptually into two different segments:

- theory - calculates clustering statistics on **simulation** volumes. Input positions are expected to be Cartesian X/Y/Z. Periodic boundary conditions are supported. Relevant C codes are in directory `theory/`
- mocks - calculates clustering statistics on **observation** volumes. Input positions are assumed to be in obverser frame, `Right Ascension`, `Declination` and `SpeedofLight*Redshift` (where required; $\omega(\theta)$ only needs `RA` and `DEC`). Relevant C codes are in directory `mocks/`

This getting-started guide assumes you have already followed the *Package Installation* section of the documentation to get the package and its dependencies set up on your machine.

If you want to compute correlation functions and have installed the python extensions, then see *Typical Tasks for Computing Correlation Functions* for typical tasks. Otherwise, read on for the various interfaces available within Corrfunc.

1.2.1 Computing Clustering Statistics with Corrfunc

Corrfunc supports three separate mechanisms to compute the clustering statistics:

- **Via python** (if you have `python` and `numpy` installed)

  Pros: Fully flexible API to modulate code behaviour at runtime. For instance, calculations can be performed in double-precision simply by passing arrays of doubles (rather than floats).

  Cons: Has fixed python overhead. For low particle numbers, can be as much as 20% slower compared to the command-line executables.

See *Using the python extensions in Corrfunc* for details on how to use the python interface.

- **Via static libraries directly in C codes**

  Pros: Fully flexible API to modulate code behaviour at runtime. All features supported by the python extensions are also supported here.

  Cons: Requires coding in C. See example C codes invoking the `theory` and `mocks` in the directories: `theory/examples/run_correlations.c` and `mocks/examples/run_correlations_mocks.c`.

  See *Using the static library interface in Corrfunc* for details on how to use the static library interface.

- **Command-line executables**

  Pros: Fastest possible implementations of all clustering statistics

  Cons: API is fixed. Any changes require full re-compilation.

  See *Using the command-line interface in Corrfunc* for details on how to use the command-line executables.

1.2.2 Available Corrfunc interfaces

Using the python extensions in Corrfunc

This guide assumes that you already followed the *Package Installation* section of the documentation to get the package and its dependencies set up on your machine. Rest of document also assumes that you have installed the C extensions for python.

Importing Corrfunc

After installing Corrfunc you can open up a python terminal and import the base package by:

```
>>> import Corrfunc
```

All of the functionality is divided into `theory` routines and `mocks` routines. These routines can be independently imported by using:

```
>>> from Corrfunc.theory import *
>>> from Corrfunc.mocks import *
```

You can access the full API documentation by simply typing:

```
help(DD)               # theory pair-counter in 3-D separation (r)
help(DDrppi_mocks)     # mocks pair-counter in 2-D (rp, pi)
```

First steps with Corrfunc

Overview of Corrfunc inputs

Broadly speaking, Corrfunc requires these following inputs:

- (At least) 3 arrays specifying the positions for the particles
  - For `Corrfunc.theory` routines, these positions are Cartesian XYZ in co-moving `Mpc/h` units.

  - For `Corrfunc.mocks` routines, these positions are `Right Ascension`, `Declination`, and `Speed of Light * Redshift` or `Co-moving distance`. The angles are expected in degrees, while the distance is expected in co-moving `Mpc/h`.

    See *Reading Catalogs for Corrfunc* for details on how to read in arrays from a file.

- A boolean flag specifying in an auto-correlation or cross-correlation is being performed. In case of cross-correlations, another set of 3 arrays **must** be passed as input. This second set of arrays typically represents randoms for `Corrfunc.mocks`.
- A file containing the bins for the clustering statistic (where relevant). Look at `theory/tests/bins` for an example of the contents of the file for spatial bins. See `mocks/tests/angular_bins` for an example containing angular bins for mocks routines. Passing a filename is the most general way of specifying bins in Corrfunc. However, you can also pass in a 1-D array for the bins.

  See *Specifying the separation bins in Corrfunc* for details on how to specify the bins as a file as well as an array

See *Typical Tasks for Computing Correlation Functions* for a broad overview of the typical tasks associated with computing correlation functions. Read on for the various pair-counters available within the python interfaces of Corrfunc.

Calculating spatial clustering statistics in simulation boxes

Corrfunc can compute a range of spatial correlation functions and the counts-in-cells. For all of these calculations a few inputs are required. The following code section sets up the default inputs that are used later on in the clustering functions:

```
>>> import numpy as np
>>> from Corrfunc.io import read_catalog

# Read the default galaxies supplied with
# Corrfunc. ~ 1 million galaxies on a 420 Mpc/h cube
>>> X, Y, Z = read_catalog()

# Specify boxsize for the XYZ arrays
>>> boxsize = 420.0

# Number of threads to use
>>> nthreads = 2

# Create the bins array
>>> rmin = 0.1
>>> rmax = 20.0
>>> nbins = 20
>>> rbins = np.logspace(np.log10(rmin), np.log10(rmax), nbins + 1)

# Specify the distance to integrate along line of sight
>>> pimax = 40.0

# Specify the max. of the cosine of the angle to the LOS for
# DD(s, mu)
>>> mu_max = 1.0

# Specify the number of linear bins in `mu`
>>> nmu_bins = 20

# Specify that an autocorrelation is wanted
>>> autocorr = 1
```

Calculating 2-D projected auto-correlation (`Corrfunc.theory.wp`)

Corrfunc can directly compute the projected auto-correlation function, $w_p(r_p)$. This calculation sets periodic boundary conditions. Randoms are calculated analytically based on the supplied boxsize. The projected separation, r_p is calculated in the X-Y plane while the line-of-sight separation, π is calculated in the Z plane. Only pairs with π separation less than π_{max} are counted.

```
from Corrfunc.theory.wp import wp
results_wp = wp(boxsize, pimax, nthreads, rbins, X, Y, Z)
```

Calculating 3-D autocorrelation (`Corrfunc.theory.xi`)

Corrfunc can also compute the 3-D auto-correlation function, $\xi(r)$. Like $w_p(r_p)$, this calculation also enforces periodic boundary conditions and an auto-correlation. Randoms are calculated analytically on the supplied boxsize.

```
from Corrfunc.theory.xi import xi
results_xi = xi(boxsize, nthreads, rbins, X, Y, Z)
```

Calculating 3-D pair-counts (`Corrfunc.theory.DD`)

Corrfunc can return the pair counts in 3-D real-space for a set of arrays. The calculation can be either auto or cross-correlation, *and* with or without periodic boundaries. The pairs are always double-counted. Additionally, if the smallest bin is `0.0` for an autocorrelation, then the self-pairs *will* be counted.

```
from Corrfunc.theory.DD import DD
results_DD = DD(autocorr, nthreads, rbins, X, Y, Z)
```

Calculating 2-D pair-counts (`Corrfunc.theory.DDrppi`)

Corrfunc can return the pair counts in 2-D real-space for a set of arrays. The calculation can be either auto or cross-correlation, *and* with or without periodic boundaries. The projected separation, r_p is calculated in the X-Y plane while the line-of-sight separation, π is calculated in the Z plane.

The pairs are always double-counted. Additionally, if the smallest bin is `0.0` for an autocorrelation, then the self-pairs *will* be counted.

```
from Corrfunc.theory.DDrppi import DDrppi
results_DDrppi = DDrppi(autocorr, nthreads, pimax, rbins, X, Y, Z, boxsize=boxsize)
```

Calculating 2-D pair-counts (`Corrfunc.theory.DDsmu`)

Corrfunc can return the pair counts in 2-D real-space for a set of arrays. The calculation can be either auto or cross-correlation, *and* with or without periodic boundaries. The spatial separation, s is calculated in 3-D while mu is the cosine of angle to the line-of-sight and is calculated assuming that the Z-axis is the line-of-sight.

$$\mathbf{s} = \mathbf{v_1} - \mathbf{v_2},$$
$$\mu = \frac{(z_1 - z_2)}{\|\mathbf{s}\|}$$

where, $\mathbf{v_1} := (x_1, y_1, z_1)$ and $\mathbf{v_2} := (x_2, y_2, z_2)$ are the vectors for the two points under consideration, and, $\|\mathbf{s}\| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$

The pairs are always double-counted. Additionally, if the smallest bin is `0.0` for an autocorrelation, then the self-pairs *will* be counted.

```
from Corrfunc.theory.DDsmu import DDsmu
results_DDsmu = DDsmu(autocorr, nthreads, rbins, mu_max, nmu_bins, X, Y, Z,␣
↪boxsize=boxsize)
```

Calculating the Counts-in-Cells (`Corrfunc.theory.vpf`)

Corrfunc can calculate the counts-in-cells statistics. The simplest example for counts-in-cells is the Void Probability Function – the probability that a sphere of a certain size contains zero galaxies.

```
from Corrfunc.theory.vpf import vpf

# Maximum radius of the sphere in Mpc/h
rmax = 10.0

# Number of bins to cover up to rmax
nbins = 10

# Number of random spheres to place
nspheres = 10000

# Max number of galaxies in sphere (must be >=1)
numpN = 6

# Random number seed (used for choosing sphere centres)
seed = 42

results_vpf = vpf(rmax, nbins, nspheres, numpN, seed, X, Y, Z)
```

Calculating clustering statistics in mock catalogs

In order to calculate clustering statistics in mock catalogs, the galaxy positions are assumed to be specified as on-sky (`Right Ascension`, `Declination`, and `speed of light * redshift`). The following code section sets up the default arrays and parameters for the actual clustering calculations:

```
import numpy as np
import Corrfunc
from os.path import dirname, abspath, join as pjoin
from Corrfunc.io import read_catalog

# Mock catalog (SDSS-North) supplied with Corrfunc
mock_catalog = pjoin(dirname(abspath(Corrfunc.__file__)), "../mocks/tests/data/",
↪"Mr19_mock_northonly.rdcz.ff")
RA, DEC, CZ = read_catalog(mock_catalog)

# Randoms catalog (SDSS-North) supplied with Corrfunc
randoms_catalog = pjoin(dirname(abspath(Corrfunc.__file__)), "../mocks/tests/data/",
↪"Mr19_randoms_northonly.rdcz.ff")
RAND_RA, RAND_DEC, RAND_CZ = read_catalog(randoms_catalog)
```

(continues on next page)

(continued from previous page)

```
# Number of threads to use
nthreads = 2

# Specify cosmology (1->LasDamas, 2->Planck)
cosmology = 1

# Create the bins array
rmin = 0.1
rmax = 20.0
nbins = 20
rbins = np.logspace(np.log10(rmin), np.log10(rmax), nbins + 1)

# Specify the distance to integrate along line of sight
pimax = 40.0

# Specify the max. of the cosine of the angle to the LOS
# for DD(s, mu)
mu_max = 1.0

# Specify the number of linear bins in `mu`
nmu_bins = 20

# Specify that an autocorrelation is wanted
autocorr = 1
```

Calculating 2-D pair counts (`Corrfunc.mocks.DDrppi_mocks`)

Corrfunc can calculate pair counts for mock catalogs. The input positions are expected to be `Right Ascension`, `Declination` and `CZ` (speed of light times redshift, in `Mpc/h`). Cosmology has to be specified since `CZ` needs to be converted into co-moving distance. If you want to calculate in arbitrary cosmology, then convert `CZ` into co-moving distance, and then pass the converted array while setting the option `is_comoving_dist=True`. The projected and line of sight separations are calculated using the following equations from Zehavi et al. 2002

$$\begin{aligned} \mathbf{s} &= \mathbf{v_1} - \mathbf{v_2}, \\ \mathbf{l} &= \frac{1}{2}\left(\mathbf{v_1} + \mathbf{v_2}\right), \\ \pi &= \left(\mathbf{s} \cdot \mathbf{l}\right) / \|\mathbf{l}\|, \\ r_p^2 &= \mathbf{s} \cdot \mathbf{s} - \pi^2 \end{aligned}$$

where, $\mathbf{v_1} := (x_1, y_1, z_1)$ and $\mathbf{v_2} := (x_2, y_2, z_2)$ are the vectors for the two points under consideration, and, $\|\mathbf{s}\| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$.

Here is the python code to call `Corrfunc.mocks.DDrppi_mocks`:

```
from Corrfunc.mocks.DDrppi_mocks import DDrppi_mocks
results_DDrppi_mocks = DDrppi_mocks(autocorr, cosmology, nthreads, pimax, rbins, RA,␣
↪DEC, CZ)
```

Calculating 2-D pair counts (`Corrfunc.mocks.DDsmu_mocks`)

Corrfunc can calculate pair counts for mock catalogs. The input positions are expected to be `Right Ascension`, `Declination` and `CZ` (speed of light times redshift, in `Mpc/h`). Cosmology has to be specified since `CZ` needs to

be converted into co-moving distance. If you want to calculate in arbitrary cosmology, then convert `CZ` into co-moving distance, and then pass the converted array while setting the option `is_comoving_dist=True`. The projected and line of sight separations are calculated using the following equations from Zehavi et al. 2002

$$\begin{aligned}\mathbf{s} &= \mathbf{v_1} - \mathbf{v_2},\\ \mathbf{l} &= \frac{1}{2}\left(\mathbf{v_1} + \mathbf{v_2}\right),\\ \mu &= \left(\mathbf{s}\cdot\mathbf{l}\right) / \left(\|\mathbf{l}\|\|\mathbf{s}\|\right)\end{aligned}$$

where, $\mathbf{v_1} := (x_1, y_1, z_1)$ and $\mathbf{v_2} := (x_2, y_2, z_2)$ are the vectors for the two points under consideration, and, $\|\mathbf{s}\| = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2 + (z_1-z_2)^2}$

Here is the python code to call `Corrfunc.mocks.DDsmu_mocks`:

```
from Corrfunc.mocks.DDsmu_mocks import DDsmu_mocks
results_DDsmu_mocks = DDsmu_mocks(autocorr, cosmology, nthreads, mu_max, nmu_bins,␣
↪rbins, RA, DEC, CZ)
```

Calculating angular pair-counts (`Corrfunc.mocks.DDtheta_mocks`)

Corrfunc can compute angular pair counts for mock catalogs. The input positions are expected to be `Right Ascension` and `Declination`. Since all calculations are in angular space, cosmology is not required.

```
from Corrfunc.mocks.DDtheta_mocks import DDtheta_mocks
results_DDtheta_mocks = DDtheta_mocks(autocorr, nthreads, rbins, RA, DEC)
```

Calculating the Counts-in-Cells (`Corrfunc.mocks.vpf_mocks`)

Corrfunc can calculate the counts-in-cells statistics. The simplest example for counts-in-cells is the Void Probability Function – the probability that a sphere of a certain size contains zero galaxies.

```
from Corrfunc.mocks.vpf_mocks import vpf_mocks

# Maximum radius of the sphere in Mpc/h
rmax = 10.0

# Number of bins to cover up to rmax
nbins = 10

# Number of random spheres to place
nspheres = 10000

# Max number of galaxies in sphere (must be >=1)
numpN = 6

# Minimum number of random points needed in a ``rmax`` sphere
# such that it is considered to be entirely within the mock
# footprint. Does not matter in this case, since we already
# have the centers for the fully enclosed spheres
threshold_ngb = 1

# File with sphere centers (centers such that spheres with size
# rmax=10 Mpc/h are completely inside the survey)
centers_file = pjoin(dirname(abspath(Corrfunc.__file__)), "../mocks/tests/data/",
↪"Mr19_centers_xyz_forVPF_rmax_10Mpc.txt")
```

(continues on next page)

(continued from previous page)

```
results_vpf_mocks = vpf_mocks(rmax, nbins, nspheres, numpN, threshold_ngb, centers_
↪file, cosmology, RA, DEC, CZ, RAND_RA, RAND_DEC, RAND_CZ)
```

See the complete reference here *Corrfunc*.

Using the static library interface in Corrfunc

This guide assumes that you already followed the *Package Installation* section of the documentation to get the package and its dependencies set up on your machine. This guide also assumes some familiarity with C coding.

This concepts in this guide are implemented in the files `theory/examples/run_correlations.c` and `mocks/examples/run_correlations_mocks.c` for simulations and mock catalogs respectively.

The basic principle of using the static libraries has the following steps:

- Include the appropriate header to get the correct function signature (at compile time)
- In your code, include call with clustering function with appropriate parameters
- Compile your code with `-I </path/to/Corrfunc/include>` flags. If you have installed Corrfunc via `pip`, then use `os.path.join(os.path.dirname(Corrfunc.__file__), ../include/)` as the include header.
- Link your code with the appropriate static library. Look in the `examples/Makefile` for the linker flags.
- Run your code

Worked out example C code for clustering statistics in simulation boxes

Common setup code for the simulation C routines

In this code section, we will setup the arrays and the overall common inputs required by the C static libraries.

```
#include "io.h"

const char file[] = {"theory/tests/data/gals_Mr19.ff"};
const char fileformat[] = {"f"};
const char binfile[] = {"theory/tests/bins"};
const double boxsize=420.0;
const double pimax=40.0;
int autocorr=1;
const int nthreads=2;

double *x1=NULL, *y1=NULL, *z1=NULL, *x2=NULL, *y2=NULL, *z2=NULL;

const int64_t ND1 = read_positions(file,fileformat,sizeof(*x1),3, &x1, &y1, &z1);
x2 = x1;
y2 = y1;
z2 = z1;
const int64_t ND2 = ND1;

struct config_options options = get_config_options();
options.verbose = 1;
options.need_avg_sep = 1;
```

(continues on next page)

(continued from previous page)

```
options.periodic = 1;
options.float_type = sizeof(*x1);
```

Calculating 2-D projected auto-correlation (`theory/wp/libcountpairs_wp.a`)

Corrfunc can directly compute the projected auto-correlation function, $w_p(r_p)$. This calculation sets periodic boundary conditions. Randoms are calculated analytically based on the supplied boxsize. The projected separation, r_p is calculated in the X-Y plane while the line-of-sight separation, π is calculated in the Z plane. Only pairs with π separation less than π_{max} are counted.

```
#include "countpairs_wp.h"

results_countpairs_wp results;
int status = countpairs_wp(ND1,x1,y1,z1,
                           boxsize,
                           nthreads,
                           binfile,
                           pimax,
                           &results,
                           &options, NULL);

if(status != EXIT_SUCCESS) {
    fprintf(stderr,"Runtime error occurred while using wp static library\n");
    return status;
}

double rlow=results.rupp[0];
for(int i=1;i<results.nbin;++i) {
    fprintf(stdout,"%e\t%e\t%e\t%e\t%12"PRIu64" \n",
                   results.wp[i],results.rpavg[i],rlow,results.rupp[i],results.
↪npairs[i]);
    rlow=results.rupp[i];
}
```

This is the generic pattern for using all of the correlation function. Look in `theory/examples/run_correlations.c` for details on how to use all of the available static libraries.

Worked out example C code for clustering statistics in mock catalogs

Corrfunc can calculate pair counts for mock catalogs. The input positions are expected to be `Right Ascension`, `Declination` and `CZ` (speed of light times redshift, in `Mpc/h`). Cosmology has to be specified since `CZ` needs to be converted into co-moving distance. If you want to calculate in arbitrary cosmology, then you have two options:

- convert `CZ` into co-moving distance, and then pass the converted array while setting `config_option.is_comoving_dist=1`.
- Add another cosmology in `utils/cosmology_params.c` in the function `init_cosmology`. Then, recompile the `Corrfunc.mocks` and pass `cosmology=integer_for_newcosmology` into the relevant functions.

Common setup code for the mocks C routines

In this code section, we will setup the arrays and the overall common inputs required by the C static libraries.

```
#include "io.h"   //for read_positions function

const char file[] = {"mocks/tests/data/Mr19_mock_northonly.rdcz.dat"};
const char fileformat[] = {"a"};     // ascii format
const char binfile[] = {"mocks/tests/bins"};
const double pimax=40.0;
int autocorr=1;
const int nthreads=2;
const int cosmology=1;  // 1->LasDamas cosmology, 2->Planck

// This computes in double-precision. Change to float for computing in float
double *ra1=NULL, *dec1=NULL, *cz1=NULL, *ra2=NULL, *dec2=NULL, *cz2=NULL;

//Read-in the data
const int64_t ND1 = read_positions(file,fileformat,sizeof(*ra1),3, &ra1, &dec1, &cz1);

ra2 = ra1;
dec2 = dec1;
cz2 = cz1;
const int64_t ND2 = ND1;

struct config_options options = get_config_options();
options.verbose=1;
options.periodic=0;
options.need_avg_sep=1;
options.float_type = sizeof(*ra1);
```

Calculating 2-D pair counts (`mocks/DDrppi_mocks/libcountpairs_rp_pi_mocks.a`)

Here is a code snippet demonstrating how to calculate $DD(r_p, \pi)$ for mock catalogs. The projected separation, r_p and line of sight separation, π are calculated using the following equations from Zehavi et al 2002:

$$\begin{aligned} \mathbf{s} &= \mathbf{v_1} - \mathbf{v_2}, \\ \mathbf{l} &= \frac{1}{2}\left(\mathbf{v_1} + \mathbf{v_2}\right), \\ \pi &= \left(\mathbf{s} \cdot \mathbf{l}\right) / \|\mathbf{l}\|, \\ r_p^2 &= \mathbf{s} \cdot \mathbf{s} - \pi^2 \end{aligned}$$

where, $\mathbf{v_1}$ and $\mathbf{v_2}$ are the vectors for the two points under consideration. Here is the C code for calling `DDrppi_mocks`:

```
#include "countpairs_rp_pi_mocks.h"

results_countpairs_mocks results;
int status = countpairs_mocks(ND1,ra1,dec1,cz1,
                              ND2,ra2,dec2,cz2,
                              nthreads,
                              autocorr,
                              binfile,
                              pimax,
                              cosmology,
                              &results,
                              &options, NULL);

const double dpi = pimax/(double)results.npibin ;
```

(continues on next page)

(continued from previous page)

```
const int npibin = results.npibin;
for(int i=1;i<results.nbin;i++) {
    const double logrp = LOG10(results.rupp[i]);
    for(int j=0;j<npibin;j++) {
        int index = i*(npibin+1) + j;
        fprintf(stdout,"%10"PRIu64" %20.8lf %20.8lf  %20.8lf \n",results.
↪npairs[index],results.rpavg[index],logrp,(j+1)*dpi);
    }
}
```

This is the generic pattern for using all of the correlation function. Look in `mocks/examples/run_correlations_mocks.c` for details on how to use all of the available static libraries.

Using the command-line interface in Corrfunc

This guide assumes that you already followed the *Package Installation* section of the documentation to get the package and its dependencies set up on your machine.

Calculating spatial clustering statistics in simulation boxes

Corrfunc can compute a range of spatial correlation functions and the counts-in-cells. The easiest way to get help on the command-line is by calling the executables without any input parameters. Here is the list of executables associated with each type of clustering statistic:

Clustering Statistic	Full path to executable
$DD(r)$	`theory/DD/DD`
$DD(r_p, \pi)$	`theory/DDrppi/DDrppi`
$w_p(r_p)$	`theory/wp/wp`
$\xi(r)$	`theory/xi/xi`
$pN(n)$	`theory/vpf/vpf`

Calculating clustering statistics in mock catalogs

The list of clustering statistics supported on mock catalogs and the associated command-line executables are:

Clustering Statistic	Full path to executable
$DD(r_p, \pi)$	`mocks/DDrppi_mocks/DDrppi_mocks`
$DD(\theta)$	`mocks/DDtheta_mocks/DDtheta_mocks`
$pN(n)$	`mocks/vpf_mocks/vpf_mocks`

Cheat-sheet for all available interfaces in Corrfunc

This guide assumes that you already followed the *Package Installation* section of the documentation to get the package and its dependencies set up on your machine. There are three available interfaces in Corrfunc

- *Using the python extensions in Corrfunc*
- *Using the static library interface in Corrfunc*. The static libraries have the form `libcount<statistic>.a`; the corresponding header file is named `count<statistic>.h`.

- *Using the command-line interface in Corrfunc*

Calculating spatial clustering statistics in simulation boxes

Corrfunc can compute a range of spatial correlation functions and the counts-in-cells. The easiest way to get help on the command-line is by calling the executables without any input parameters. Here is the list of executables associated with each type of clustering statistic:

Clustering Statistic	Python Interface	Static library	Command-line (executable name)
$\xi(r)$	*Corrfunc.theory.DD*	`theory/DD/libcountpairs.a`	`theory/DD/DD`
$\xi(r_p, \pi)$	*Corrfunc.theory.DDrppi*	`theory/DDrppi/libcountpairs_rp_pi.a`	`theory/DDrppi/DDrppi`
$\xi(s, \mu)$	*Corrfunc.theory.DDsmu*	`theory/DDsmu/libcountpairs_s_mu.a`	`theory/DDsmu/DDsmu`
$w_p(r_p)$	*Corrfunc.theory.wp*	`theory/wp/libcountpairs_wp.a`	`theory/wp/wp`
$\xi(r)$	*Corrfunc.theory.xi*	`theory/xi/libcountpairs_xi.a`	`theory/xi/xi`
$pN(n)$	*Corrfunc.theory.vpf*	`theory/vpf/libcountspheres.a`	`theory/vpf/vpf`

Calculating clustering statistics in mock catalogs

The list of clustering statistics supported on mock catalogs and the associated command-line executables are:

Clustering Statistic	Python Interface	Static library	Command-line (executable name)
$\xi(r_p, \pi)$	*Corrfunc.mocks.DDrppi_mocks*	`mocks/DDrppi_mocks/libcountpairs_rp_pi_mocks.a`	`mocks/DDrppi_mocks/DDrppi_mocks`
$\xi(s, \mu)$	*Corrfunc.mocks.DDsmu_mocks*	`mocks/DDsmu_mocks/libcountpairs_s_mu_mocks.a`	`mocks/DDsmu_mocks/DDsmu_mocks`
$\omega(\theta)$	*Corrfunc.mocks.DDtheta_mocks*	`mocks/DDtheta_mocks/libcountpairs_theta_mocks.a`	`mocks/DDtheta_mocks/DDtheta_mocks`
$pN(n)$	*Corrfunc.mocks.vpf_mocks*	`mocks/vpf_mocks/libcountspheres_mocks`	`mocks/vpf_mocks/vpf_mocks`

If you are not sure which correlation function to use, then please also see *Which correlation function to use?*.

1.3 Typical Tasks for Computing Correlation Functions

Here we present docstrings of the most commonly used functions and classes grouped together by functionality. Many docstrings contain example code to demonstrate basic usage. For documentation of functions not listed here, see *Corrfunc*.

1.3.1 Reading input data

Reading Catalogs for Corrfunc

All of the `Corrfunc` routines require some sort of position arrays, X/Y/Z, as input. These arrays are expected to be 1-D arrays of type `np.array`. If you already have have the required `numpy` arrays, then you can just pass them straight to `Corrfunc`. If you need to read the arrays in from disk, then read on. For the command-line interface, the input files can only be in ASCII or fast-food format (for description of fast-food binaries, see *Fast-food binary format*).

Fast-food binary 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 file. The rest can simply filled with *0*.

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

Note: Even when the `X/Y/Z` arrays are written out in double-precision, the padding is still 4 bytes. The blocks for `Y/Z` similarly follow after the `X` block.

Reading from ASCII files

This is the most straight forward way – you need an ASCII file with columns X/Y/Z (white-space separated).

Using `numpy.genfromtxt`

```
import numpy as np
fname = "myfile_containing_xyz_columns.dat"

# For double precision calculations
dtype = np.float  ## change to np.float32 for single precision

X, Y, Z = np.genfromtxt(fname, dtype=dtype, unpack=True)
```

Note: `Corrfunc.read_catalog` uses this exact code-snippet to read in ASCII files in python.

Reading from fast-food files

If you are using the command-line interface, then the code will **have** to read the arrays from files. While `Corrfunc` natively supports both ASCII and fast-food formats (for description of fast-food binaries, see *Fast-food binary format*), the following python utility is intended to read both these types of files.

Using utility: `Corrfunc.io.read_catalog`

Corrfunc.io.read_catalog can directly read ASCII files or fast-food binary files.

```
from Corrfunc.io import read_catalog

# Read the standard theory catalog (on a box)
# supplied with Corrfunc
X, Y, Z = read_catalog()

# Read some other format -> have to specify
# filename
fname = "myfile_containing_xyz_columns.dat"
X, Y, Z = read_catalog(fname)
```

1.3.2 Creating a file with bins for the clustering statistics

Specifying the separation bins in Corrfunc

All of the python extensions for *Corrfunc* accept either a filename or an array for specifying the r_p or θ.

Manually creating a file with arbitrary bins

This manual method lets you specify generic bins as long as the upper-edge of one bin is the same as the lower-edge of the next (i.e., continuous bins). The bins themselves can have arbitrary widths, and the smallest bin can start from 0.0.

- Open a text editor with a new file
- Add two columns per bin you want, the first column should be low-edge of the bin while the second column should be the high-edge of the bin. Like so:

```
0.10     0.15
```

- Now add as many such lines as the number of bins you want. Here is a valid example:

```
0.10     0.15
0.15     0.50
0.50     5.00
```

This example specifies 3 bins, with the individual bin limits specified on each line. Notice that the width of each bin can be independently specified (but the bins **do** have to be continuous)

Note: Make sure that the bins are in increasing order – smallest bin first, then the next smallest bin and so on up to the largest bin.

Specifying bins as an array

You can specify the bins using `numpy.linspace` or `numpy.logspace`.

```
import numpy as np
rmin = 0.1
rmax = 10.0
nbins = 20
rbins = np.linspace(rmin, rmax, nbins + 1)
log_rbins = np.logspace(np.log10(rmin), np.log10(rmax), nbins + 1)
```

1.3.3 Choosing the correlation function

Which correlation function to use?

Corrfunc has a variety of correlation functions to cover a broad range of Science applications. The basic distinction occurs if the input particles are directly from a simulation or from an observational survey (or equivalently, a simulation that has been processed to look like a survey). For simulation data, referred throughout as *theory*, the assumption is that the particle positions are Cartesian, co-moving XYZ. For survey data, referred throughout as *mocks*, the assumption is that particle positions are *Right Ascension* (0 – 360 deg), *Declination* (-90 – 90 deg) and *CZ* (speed of light multiplied by the redshift). Depending on the exact type of data, **and** the desired correlation function you want, the following table should help you figure out which code you should use.

Input Data	Periodic	Particle domain	Desired correlation function	Returns	Python code
X, Y, Z	True	Cube (box)	wp(r_p)	2-D Projected Correlation	*Corrfunc.theory.wp*
			$\xi(r)$	3-D Real-space Correlation	*Corrfunc.theory.xi*
X, Y, Z	True or False	Arbitrary	$\xi(r)$	Pair-counts in 3-D real-space	*Corrfunc.theory.DD*
			$\xi(r_p, \pi)$	Pair-counts in 2-D	*Corrfunc.theory.DDrppi*
			$\xi(s, \mu)$	Pair-counts in 2-D	*Corrfunc.theory.DDsmu*
ra, dec, cz	False	Arbitrary	$\xi(r_p, \pi)$	Pair-counts in 2-D	*Corrfunc.mocks.DDrppi_mocks*
			$\xi(s, \mu)$	Pair-counts in 2-D	*Corrfunc.mocks.DDsmu_mocks*
ra, dec	False	Arbitrary	$\omega(\theta)$	Pair-counts in angular space	*Corrfunc.mocks.DDtheta_mocks*

In all cases where only pair-counts are returned (e.g., all of the *mocks* routines), you will need to compute at least an additional *RR* term. Please see *Corrfunc.utils.convert_3d_counts_to_cf* to convert 3-D pair-counts (or angular pair counts) into a correlation function. For 2-D pair-counts, please use *Corrfunc.utils.convert_rp_pi_counts_to_wp* to convert into a projected correlation function. If you want to compute the $\xi(r_p, \pi)$ from the 2-D pair-counts, then simply call *Corrfunc.utils.convert_3d_counts_to_cf* with the arrays.

Also, see *Using the command-line interface in Corrfunc* for a detailed list of the clustering statistics and the various available API interfaces.

1.3.4 Calculating Correlation Functions on Simulations

Converting 3D pair counts into a correlation function

3D pair counts can be converted into a correlation function by using the helper function *Corrfunc.utils.convert_3d_counts_to_cf*. First, we have to compute the relevant pair counts using the python wrapper *Corrfunc.theory.DD*

```
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> from Corrfunc.theory.DD import DD
>>> from Corrfunc.io import read_catalog
>>> from Corrfunc.utils import convert_3d_counts_to_cf

>>> # Read the supplied galaxies on a periodic box
>>> X, Y, Z = read_catalog()
>>> N = len(X)
>>> boxsize = 420.0
>>> nthreads = 2

# Generate randoms on the box
>>> rand_N = 3*N
>>> rand_X = np.random.uniform(0, boxsize, rand_N)
>>> rand_Y = np.random.uniform(0, boxsize, rand_N)
>>> rand_Z = np.random.uniform(0, boxsize, rand_N)

# Setup the bins
>>> nbins = 10
>>> bins = np.linspace(0.1, 10.0, nbins + 1) # note that +1 to nbins

# Auto pair counts in DD
>>> autocorr=1
>>> DD_counts = DD(autocorr, nthreads, bins, X, Y, Z,
...                periodic=False, verbose=True)

# Cross pair counts in DR
>>> autocorr=0
>>> DR_counts = DD(autocorr, nthreads, bins, X, Y, Z,
...                X2=rand_X, Y2=rand_Y, Z2=rand_Z,
...                periodic=False, verbose=True)

# Auto pairs counts in RR
>>> autocorr=1
>>> RR_counts = DD(autocorr, nthreads, bins, rand_X, rand_Y, rand_Z,
...                periodic=False, verbose=True)

# All the pair counts are done, get the correlation function
>>> cf = convert_3d_counts_to_cf(N, N, rand_N, rand_N,
...                              DD_counts, DR_counts,
...                              DR_counts, RR_counts)
```

See the complete reference here *Corrfunc*.

Converting (r_p, π) pairs into a projected correlation function

Pair counts in (r_p, π) can be converted into a projected correlation function by using the helper function *Corrfunc.utils.convert_rp_pi_counts_to_wp*.

```
>>> import numpy as np
>>> from Corrfunc.theory import DDrppi
>>> from Corrfunc.io import read_catalog
>>> from Corrfunc.utils import convert_rp_pi_counts_to_wp

# Read the supplied galaxies on a periodic box
>>> X, Y, Z = read_catalog()
>>> N = len(X)
>>> boxsize = 420.0

# Generate randoms on the box
>>> rand_N = 3*N
>>> rand_X = np.random.uniform(0, boxsize, rand_N)
>>> rand_Y = np.random.uniform(0, boxsize, rand_N)
>>> rand_Z = np.random.uniform(0, boxsize, rand_N)
>>> nthreads = 2
>>> pimax = 40.0

# Setup the bins
>>> nrpbins = 10
>>> bins = np.linspace(0.1, 10.0, nrpbins + 1)

# Auto pair counts in DD
>>> autocorr=1
>>> DD_counts = DDrppi(autocorr, nthreads, bins, X, Y, Z,
...                    periodic=False, verbose=True)

# Cross pair counts in DR
>>> autocorr=0
>>> DR_counts = DDrppi(autocorr, nthreads, bins, X, Y, Z,
...                    X2=rand_X, Y2=rand_Y, Z2=rand_Z,
...                    periodic=False, verbose=True)

# Auto pairs counts in RR
>>> autocorr=1
>>> RR_counts = DDrppi(autocorr, nthreads, bins, rand_X, rand_Y, rand_Z,
...                    periodic=False, verbose=True)

# All the pair counts are done, get the correlation function
>>> wp = convert_rp_pi_counts_to_wp(N, N, rand_N, rand_N,
...                                 DD_counts, DR_counts,
...                                 DR_counts, RR_counts, nrpbins, pimax)
```

See the complete reference here *Corrfunc*.

Directly Computing $\xi(r)$ and $wp(rp)$

For a periodic cosmological box, the 3-d auto correlation, $\xi(r)$, and the projected auto correlation function, $wp(rp)$, can be directly computed using the Natural Estimator. The relevant python wrappers are present in *Corrfunc.theory.xi* and *Corrfunc.theory.wp*. See *Notes on the Random-Random Term in Autocorrelations* for details on how the Natural Estimator is computed.

```
>>> import numpy as np
>>> from Corrfunc.theory.wp import wp
>>> from Corrfunc.theory.xi import xi
>>> from Corrfunc.io import read_catalog
```

(continues on next page)

(continued from previous page)

```
>>> X, Y, Z = read_catalog()
>>> boxsize = 420.0
>>> nthreads = 2
>>> pimax = 40.0
>>> nbins = 10
>>> bins = np.linspace(0.1, 10.0, nbins + 1) # Note the + 1 to nbins
>>> wp_counts = wp(boxsize, pimax, nthreads, bins, X, Y, Z)
>>> xi_counts = xi(boxsize, nthreads, bins, X, Y, Z)
```

See the complete reference here *Corrfunc*.

Detailed API for Clustering Statistics on Simulations

All of these can be imported from *Corrfunc.theory*. See the complete reference here *Corrfunc*.

Clustering in 3-D

- Pair counts for (auto or cross) correlations for $\xi(r)$ – *Corrfunc.theory.DD*
- Auto-correlation on periodic, cosmological boxes, $\xi(r)$, – *Corrfunc.theory.xi*

Clustering in 2-D

- Pair counts (auto or cross) correlations for $\xi(rp, \pi)$ – *Corrfunc.theory.DDrppi*
- Pair counts (auto or cross) correlations for $\xi(s, \mu)$ – *Corrfunc.theory.DDsmu*
- Projected auto-correlation function, $wp(rp)$ – *Corrfunc.theory.wp*

Counts-in-cells

- Void Probability functions and counts-in-cells stats $pN(r)$ – *Corrfunc.theory.vpf*

Notes on the Random-Random Term in Autocorrelations

The following discussion is adapted from this notebook by Lehman Garrison.

When computing a two-point correlation function estimator like

$$\xi(r) = \quad - 1,$$

the term can be computed analytically if the domain is a periodic box. Often, this is done as

$$\begin{aligned} {}_i &= NV_i\bar{\rho} \\ &= NV_i\frac{N}{L^3} \end{aligned} \tag{1.1}$$

where ${}_i$ is the expected number of random-random pairs in bin i, N is the total number of points, V_i is the volume (or area if 2D) of bin i, L is the box size, and $\bar{\rho}$ is the average density in the box.

However, using $\bar{\rho} = \frac{N}{L^3}$ is only correct for continuous fields, not sets of particles. When sitting on a particle, only $N-1$ particles are available to be in a bin at some non-zero distance. The remaining particle is the particle you're sitting on, which is always at distance 0. Thus, the correct expression is

$$_i = NV_i \frac{N-1}{L^3}.$$

See this notebook for an empirical demonstration of this effect; specifically, that computing the density with $N-1$ is correct, and that using N introduces bias of order $\frac{1}{N}$ into the estimator. This is a tiny correction for large N problems, but important for small N.

Any `Corrfunc` function that returns a clustering statistic (not just raw pair counts) implements this correction. Currently, this includes *`Corrfunc.theory.xi`* and *`Corrfunc.theory.wp`*.

Cross-correlations of two different particle sets don't suffer from this problem; the particle you're sitting on is never part of the set of particles under consideration for pair-making.

`Corrfunc` also allows bins of zero separation, in which "self-pairs" are included in the pair counting. $_i$ must reflect this by simply adding N to any such bin.

RR in Weighted Clustering Statistics

We can extend the above discussion to weighted correlation functions in which each particle is assigned a weight, and the pair weight is taken as the product of the particle weights (see *Computing Weighted Correlation Functions*).

Let w_j be the weight of particle j, and W be the sum of the weights. We will define the "unclustered" particle distribution to be the case of N particles uniformly distributed, where each is assigned the mean weight $\bar{w}$. We thus have

$$\begin{aligned} _i &= \sum_{j=1}^{N} \bar{w}(W - \bar{w}) \frac{V_i}{L^3} \\ &= (W^2 - \bar{w}W) \frac{V_i}{L^3} \\ &= W^2 \left(1 - \frac{1}{N}\right) \frac{V_i}{L^3} \end{aligned} \tag{1.3}$$

When the particles all have $w_j = 1$, then $W = N$ and we recover the unweighted result from above.

There are other ways to define the unclustered distribution. If we were to redistribute the particles uniformly but preserve their individual weights, we would find

$$\begin{aligned} _i &= \sum_{j=1}^{N} w_j(W - w_j) \frac{V_i}{L^3} \\ &= \left(W^2 - \sum_{j=1}^{N} w_j^2\right) \frac{V_i}{L^3} \end{aligned} \tag{1.6}$$

This is not what we use in `Corrfunc`, but this should help illuminate some of the considerations that go into defining the "unclustered" case when writing a custom weight function (see *Implementing Custom Weight Functions*).

1.3.5 Calculating Correlation Functions on Mock Catalogs

Calculating the projected correlation function, $wp(rp)$

2-D Pair counts can be converted into a $wp(rp)$ by using the helper function *`Corrfunc.utils.convert_rp_pi_counts_to_wp`*. First, we have to compute the relevant pair counts using the python wrapper *`Corrfunc.mocks.DDrppi_mocks`*

```
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.mocks.DDrppi_mocks import DDrppi_mocks
>>> from Corrfunc.io import read_catalog
>>> from Corrfunc.utils import convert_rp_pi_counts_to_wp

>>> galaxy_catalog=pjoin(dirname(abspath(Corrfunc.__file__)),
...                      "../mocks/tests/data", "Mr19_mock_northonly.rdcz.ff")

# Read the supplied galaxies on a periodic box
>>> RA, DEC, CZ = read_catalog(galaxy_catalog)
>>> N = len(RA)

# Read the supplied randoms catalog
>>> random_catalog=pjoin(dirname(abspath(Corrfunc.__file__)),
...                      "../mocks/tests/data", "Mr19_randoms_northonly.rdcz.ff")
>>> rand_RA, rand_DEC, rand_CZ = read_catalog(random_catalog)
>>> rand_N = len(rand_RA)

# Setup the bins
>>> nbins = 10
>>> bins = np.linspace(0.1, 20.0, nbins + 1)
>>> pimax = 40.0

>>> cosmology = 1
>>> nthreads = 2

# Auto pair counts in DD
>>> autocorr=1
>>> DD_counts = DDrppi_mocks(autocorr, cosmology, nthreads, pimax, bins,
...                          RA, DEC, CZ)


# Cross pair counts in DR
>>> autocorr=0
>>> DR_counts = DDrppi_mocks(autocorr, cosmology, nthreads, pimax, bins,
...                          RA, DEC, CZ,
...                          RA2=rand_RA, DEC2=rand_DEC, CZ2=rand_CZ)


# Auto pairs counts in RR
>>> autocorr=1
>>> RR_counts = DDrppi_mocks(autocorr, cosmology, nthreads, pimax, bins,
...                          rand_RA, rand_DEC, rand_CZ)

# All the pair counts are done, get the angular correlation function
>>> wp = convert_rp_pi_counts_to_wp(N, N, rand_N, rand_N,
...                                 DD_counts, DR_counts,
...                                 DR_counts, RR_counts, nbins, pimax)
```

See the complete reference here *Corrfunc*.

Calculating the angular correlation function, $\omega(\theta)$

Angular pair counts can be converted into a $\omega(\theta)$ by using the helper function *Corrfunc.utils.convert_3d_counts_to_cf*. First, we have to compute the relevant pair counts using the python wrapper

Corrfunc.mocks.DDtheta_mocks

```
>>> from os.path import dirname, abspath, join as pjoin
>>> import numpy as np
>>> import Corrfunc
>>> from Corrfunc.mocks.DDtheta_mocks import DDtheta_mocks
>>> from Corrfunc.io import read_catalog
>>> from Corrfunc.utils import convert_3d_counts_to_cf

>>> galaxy_catalog=pjoin(dirname(abspath(Corrfunc.__file__)),
...                      "../mocks/tests/data",
...                      "Mr19_mock_northonly.rdcz.ff")

# Read the supplied galaxies on a periodic box
>>> RA, DEC, _ = read_catalog(galaxy_catalog)

# Read the supplied randoms catalog
>>> random_catalog=pjoin(dirname(abspath(Corrfunc.__file__)),
...                      "../mocks/tests/data", "Mr19_randoms_northonly.rdcz.ff")
>>> rand_RA, rand_DEC, _ = read_catalog(random_catalog)
>>> rand_N = len(rand_RA)

# Setup the bins
>>> nbins = 10
>>> bins = np.linspace(0.1, 10.0, nbins + 1) # note the +1 to nbins

# Number of threads to use
>>> nthreads = 2

# Auto pair counts in DD
>>> autocorr=1
>>> DD_counts = DDtheta_mocks(autocorr, nthreads, bins,
...                           RA, DEC)

# Cross pair counts in DR
>>> autocorr=0
>>> DR_counts = DDtheta_mocks(autocorr, nthreads, bins,
...                           RA, DEC,
...                           RA2=rand_RA, DEC2=rand_DEC)

# Auto pairs counts in RR
>>> autocorr=1
>>> RR_counts = DDtheta_mocks(autocorr, nthreads, bins,
...                           rand_RA, rand_DEC)

# All the pair counts are done, get the angular correlation function
>>> wtheta = convert_3d_counts_to_cf(N, N, rand_N, rand_N,
...                                  DD_counts, DR_counts,
...                                  DR_counts, RR_counts)
```

See the complete reference here *Corrfunc*.

Detailed API for Clustering Statistics on Mock Catalogs

All of these can be imported from *Corrfunc.mocks*. See the complete reference here *Corrfunc*.‘

Clustering in 2-D

- Pair counts (auto or cross) correlations for $\xi(rp, \pi)$ – *Corrfunc.mocks.DDrppi_mocks*
- Pair counts (auto or cross) correlations for $\xi(s, \mu)$ – *Corrfunc.mocks.DDsmu_mocks*

Angular clustering

- Pair counts (auto or cross) correlations for $\omega(\theta)$ – *Corrfunc.mocks.DDtheta_mocks*

Counts-in-cells

- Void Probability functions and counts-in-cells stats $pN(r)$ – *Corrfunc.mocks.vpf_mocks*

1.3.6 Weighted Correlation Functions

Computing Weighted Correlation Functions

Every clustering statistic in `Corrfunc` accepts an array of weights that can be used to compute weighted correlation functions. The API reference for each clustering statistic (*Corrfunc.theory.xi*, *Corrfunc.mocks.DDrppi_mocks*, etc.) contains examples of how to do this. The interface is standard across functions: the inputs are a `weights` array and a `weight_type` string that specifies how to use the "point weights" to compute a "pair weight". Currently, the only supported `weight_type` is `pair_product`, in which the pair weight is the product of the point weights (but see *Implementing Custom Weight Functions* for how to write your own function).

If `weight_type` and `weights` (or `weights1` and `weights2` for cross-correlations) are given, the mean pair weight in a separation bin will be given in the `weightavg` field of the output. This field is 0.0 if weights are disabled.

Pair counts (i.e. the `npairs` field in the `results` array) are never affected by weights. For theory functions like *Corrfunc.theory.xi* and *Corrfunc.theory.wp* that actually return a clustering statistic, the statistic is weighted. For `pair_product`, the distribution used to compute the expected bin weight from an unclustered particle set (the `RR` term) is taken to be a spatially uniform particle set where every particle has the mean weight. See *RR in Weighted Clustering Statistics* for more discussion.

Running with weights incurrs a modest performance hit (around 20%, similar to enabling `ravg`). Weights are supported for all instruction sets (SSE, AVX, and fallback).

Consider the following simple example adapted from the *Corrfunc.theory.xi* docstring, in which we assign a weight of 0.5 to every particle and get the expected average pair weight of 0.25 (last column of the output). Note that `xi` (fourth column) is also weighted, but the case of uniform weights is equivalent to the unweighted case.

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.theory.xi import xi
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../theory/tests/", "bins")
>>> N = 100000
>>> boxsize = 420.0
>>> nthreads = 4
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(0, boxsize, N)
```

(continues on next page)

(continued from previous page)

```
>>> Y = np.random.uniform(0, boxsize, N)
>>> Z = np.random.uniform(0, boxsize, N)
>>> weights = np.full_like(X, 0.5)
>>> results = xi(boxsize, nthreads, binfile, X, Y, Z, weights=weights, weight_type=
↪'pair_product', output_ravg=True)
>>> for r in results: print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10.6f} {4:10d} {5:10.6f}
↪"
...                          .format(r['rmin'], r['rmax'],
...                          r['ravg'], r['xi'], r['npairs'], r['weightavg']))
...
      0.167536   0.238755   0.226592  -0.205733          4   0.250000
      0.238755   0.340251   0.289277  -0.176729         12   0.250000
      0.340251   0.484892   0.426819  -0.051829         40   0.250000
      0.484892   0.691021   0.596187  -0.131853        106   0.250000
      0.691021   0.984777   0.850100  -0.049207        336   0.250000
      0.984777   1.403410   1.225112   0.028543       1052   0.250000
      1.403410   2.000000   1.737153   0.011403       2994   0.250000
      2.000000   2.850200   2.474588   0.005405       8614   0.250000
      2.850200   4.061840   3.532018  -0.014098      24448   0.250000
      4.061840   5.788530   5.022241  -0.010784      70996   0.250000
      5.788530   8.249250   7.160648  -0.001588     207392   0.250000
      8.249250  11.756000  10.207213  -0.000323     601002   0.250000
     11.756000  16.753600  14.541171   0.000007    1740084   0.250000
     16.753600  23.875500  20.728773  -0.001595    5028058   0.250000
```

Implementing Custom Weight Functions

`Corrfunc` supports custom weight functions. On this page we describe the recommended procedure for writing your own. When in doubt, follow the example of `pair_product`.

First, see *Computing Weighted Correlation Functions* for basic usage of `Corrfunc`'s weight features.

The steps are:

1. Add a type to the `weight_method_t` enum in `utils/defs.h` (something like `MY_WEIGHT_SCHEME=1`).
2. Determine how many weights per particle your scheme needs, and add a case to the switch-case block in `get_num_weights_by_method()` in `utils/defs.h`. `Corrfunc` supports up to `MAX_NUM_WEIGHTS=10` weights per particle; most schemes will simply need 1. To provide multiple weights per particle via the Python interface, simply pass a `weights` array of shape `(N_WEIGHTS_PER_PARTICLE, N_PARTICLES)`.
3. Add an `if` statement that maps a string name (like "my_weight_scheme") to the `weight_method_t` (which you created above) in `get_weight_method_by_name()` in `utils/defs.h`.
4. Write a function in `utils/weight_functions.h.src` that returns the weight for a particle pair, given the weights for the two particles. The weights for each particle are packed in a `const pair_struct_DOUBLE` struct, which also contains the pair separation. You must write one function for every instruction set you wish to support. This can be quite easy for simple weight schemes; the three functions for `pair_product` are:

```
/*
 * The pair weight is the product of the particle weights
 */
static inline DOUBLE pair_product_DOUBLE(const pair_struct_DOUBLE *pair){
    return pair->weights0[0].d*pair->weights1[0].d;
```

(continues on next page)

(continued from previous page)

```
}

#ifdef __AVX__
static inline AVX_FLOATS avx_pair_product_DOUBLE(const pair_struct_DOUBLE *pair){
    return AVX_MULTIPLY_FLOATS(pair->weights0[0].a, pair->weights1[0].a);
}
#endif

#ifdef __SSE4_2__
static inline SSE_FLOATS sse_pair_product_DOUBLE(const pair_struct_DOUBLE *pair){
    return SSE_MULTIPLY_FLOATS(pair->weights0[0].s, pair->weights1[0].s);
}
#endif
```

See `utils/avx_calls.h` and `utils/sse_calls.h` for the lists of available vector instructions.

5. For each function you wrote in the last step, add a case to the switch-case block in the appropriate dispatch function in `utils/weight_functions.h.src`. If you wrote a weighting function for all three instruction sets, then you'll need to add the corresponding function to `get_weight_func_by_method_DOUBLE()`, `get_avx_weight_func_by_method_DOUBLE()`, and `get_sse_weight_func_by_method_DOUBLE()`.

6. Done! Your weight scheme should now be accessible through the Python and C interfaces via the name ("my_weight_scheme") that you specified above. The output will be accessible in the `weightavg` field of the `results` array.

Pair counts (i.e. the `npairs` field in the `results` array) are never affected by weights. For theory functions like *`Corrfunc.theory.xi`* and *`Corrfunc.theory.wp`* that actually return a clustering statistic, the statistic is weighted. For `pair_product`, the random distribution used to compute the expected bin weight from an unclustered particle set (the `RR` term) is taken to be a spatially uniform particle set where every particle has the mean weight. See *RR in Weighted Clustering Statistics* for more discussion. This behavior (automatically returning weighted clustering statistics) is only implemented for `pair_product`, since that is the only weighting method for which we know the desired equivalent random distribution. Custom weighting methods can implement similar behavior by modifying `countpairs_xi_DOUBLE()` in `theory/xi/countpairs_xi_impl.c.src` and `countpairs_wp_DOUBLE()` in `theory/wp/countpairs_wp_impl.c.src`.

1.4 Developer documentation

The developer documentation contains guidlines for how to stay up-to-date on Corrfunc development, submit bug reports and contribute to the Corrfunc code base.

1.4.1 License and Citation Information

Citing Corrfunc

Corrfunc is currently preparing for its first "official" release (v2.0.0). The v2.0.0 release is accompanied with a code-release paper, ArXiv. If you use Corrfunc modules in your analysis, please cite this code-release paper. While the paper is being written, you can cite Corrfunc with the Zenodo DOI:

```
@misc{manodeep_sinha_2016_55161,
author       = {Manodeep Sinha},
title        = {Corrfunc: Corrfunc-1.1.0},
```

(continues on next page)

(continued from previous page)

```
month        = jun,
year         = 2016,
doi          = {10.5281/zenodo.55161},
url          = {{http://dx.doi.org/10.5281/zenodo.55161}}
```

Corrfunc License

Corrfunc comes with a MIT LICENSE - see the LICENSE file.

1.4.2 Package contributors

Corrfunc project coordinator

- Manodeep Sinha

Lead developers

- Manodeep Sinha

Core package contributors

- Manodeep Sinha (@manodeep)
- Lehman Garrison (@lgarrison)
- Nick Hand (@nickhand)

Other credits

- Corrfunc contains code from Agner Fog, GeometricTools, and the package SGLIB. The LICENSE for these external files remains with the original author of the package.
- The entirety of the docs for Corrfunc is derived from halotools. I know, first-hand, how much of an effort it was for the developers of *halotools* to generate all of this documentation. Having such a template made creating the docs for Corrfunc a lot easier process.

- The API generation script for Corrfunc was lifted directly out of the repo bccp/nbodykit/.

1.4.3 Submitting a Bug Report

If you find or just suspect buggy behavior in Corrfunc, please raise an issue on GitHub. Navigate to the Corrfunc Issues page, create a new issue with a description of the problem and the full Traceback (if applicable), and attach a *bug* label to the issue.

1.4.4 Staying Up to Date

If you would like to receive notifications of new code releases, sign up for the google group

https://groups.google.com/forum/#!forum/Corrfunc

Feel free to ask questions about the code on the group. However, note that all exchanges on the groups are subject to Astropy Community Code of Conduct, which is basically, "Be nice!". If you are unsure about some technical aspect of the code, then feel free to email the author (Manodeep Sinha).

1.4.5 Contributing to Corrfunc

Corrfunc is written in a very modular fashion with minimal interaction between the various calculations. The algorithm presented in Corrfunc is applicable to a broad-range of astrophysical problems, viz., any situation that requires looking at *all* objects around a target and performing some analysis with this group of objects.

Here are the basic steps to get your statistic into the Corrfunc package:

- Fork the repo and add your statistic
- Add exhaustive tests. The output of your statistic should **exactly** agree with a brute-force implementation (under double-precision). Look at `test_periodic.c` and `test_nonperiodic.c` under `theory/tests/` for tests on simulation volumes. For mock catalogs, look at `mocks/tests/tests_mocks.c`.
- Add a python extension for the new statistic. This extension should reside in file `theory/python_bindings/_countpairs.c` or `mocks/python_bindings/_countpairs_mocks.c` for statistics relevant for simulations and mocks respectively. It is preferred to have the extension documented but not necessary.
- Add a call to this new *extension* in the `python_bindings/call_correlation_functions*.py` script.

Note: Different from corresponding script in `Corrfunc/` directory.

- Add a python wrapper for the previous python extension. This wrapper should exist in `Corrfunc/theory/` or `Corrfunc/mocks/`. Wrapper **must** have inline API docs.
- Add the new wrapper to `__all__` in `__init__.py` within the relevant directory.
- Add an example call to this *wrapper* in `Corrfunc/call_correlation_functions.py` or `Corrfunc/call_correlation_functions_mocks.py` for simulations and mocks respectively.

Note: Different from corresponding script in `python_bindings` directory.

- Add the new wrapper to the API docs within `ROOT_DIR/docs/source/theory_functions.rst` or `ROOT_DIR/docs/source/mocks_functions.rst`.

- Add to the contributors list under `ROOT_DIR/docs/source/development/contributors.rst`.
- Submit pull request

Note: Please feel free to email the author or the Corrfunc Google Groups if you need help at any stage.

Corrfunc Design

All of the algorithms in Corrfunc have the following components:

- Reading in data. Relevant routines are in the `io/` directory with a mapping within `io.c` to handle the file format
- Creating the 3-D lattice structure. Relevant routines are in the `utils/gridlink_impl.c.src` and `utils/gridlink_mocks.c.src`. This lattice grids up the particle distribution on cell-sizes of `rmax` (the maximum search radius).

Note: The current lattice code duplicates the particle memory. If you need a lattice that does not duplicate the particle memory, then please email the author. Relevant code existed in Corrfunc but has been removed in the current incarnation.

- Setting up the OpenMP sections such that threads have local copies of histogram arrays. If OpenMP is not enabled, then this section should not produce any compilable code.
- Looping over all cells in the 3-D lattice and then looping over all neighbouring cells for each cell.
- For a pair of cells, hand over the two sets of arrays into a specialized kernel (`count*kernel.c.src`) for computing pairs.
- Aggregate the results, if OpenMP was enabled.

Directory and file layout

- Codes that compute statistics on simulation volumes (Cartesian XYZ as input) go into a separate directory within `theory`
- Codes that compute statistics on mock catalogs (RA, DEC [CZ]) go into a separate directory within `mocks`
- Public API in a `count*.h` file. Corresponding C file simply dispatches to appropriate floating point implementation.
- Floating point implmentation in file `count*_impl.c.src`. This file is processed via `sed` to generate both single and double precision implementations.
- A kernel named `count*kernels.c.src` containing implementations for counting pairs on two sets of arrays. This kernel file is also preprocessed to produce both the single and double precision kernels.
- Tests go within `tests` directory under `theory` or `mocks`, as appropriate. For simulation routines, tests with and without periodic boundaries go into `test_periodic.c` and `test_nonperiodic.c`
- C code to generate the python extensions goes under `python_bindings` directory into the file `_countpairs*.c`
- Each python extension has a python wrapper within `Corrfunc` directory

Coding Guidelines

C guidelines

Code contents

- **Always** check for error conditions when calling a function
- If an error condition occurs when making an kernel/external library call, first call `perror` and then return the error status. If calling a wrapper from within Corrfunc, assume that `perror` has already been called and simply return the status. Clean up memory before returning status.
- Declare variables in the smallest possible scope.
- Add `const` qualifiers liberally
- There **must** not be any compiler warnings (with `gcc6.0`) under the given set of Warnings already enabled within `common.mk`. If the warning can not be avoided because of logic issues, then suppress the warning but note why that suppression is required. Warnings are treated as errors on the continuous integration platform (TRAVIS)
- Valgrind should not report any fixable memory or file leaks (memory leaks in OpenMP library, e.g., `libgomp`, are fine)

Style

The coding style is loosely based on Linux Kernel Guideline. These are recommended but not strictly enforced. However, note that if you do contribute code to Corrfunc, the style may get converted.

- Braces - Opening braces start at the same line, except for functions - Closing braces on new line - Even single line conditionals must have opening and closing braces
- Comments - Explanatory comments on top of code segment enclosed with `/**/` - Inline comments must be single-line on the right
- Indentation is `tab:=4 spaces`
- Avoid `typedef` for `structs` and `unions`

Python guidelines

- Follow the astropy python code guide
- Docs are in `numpydocs` format. Follow any of the wrapper routines in `Corrfunc` (which are, in turn, taken from halotools)

CHAPTER 2

Reference

2.1 Comprehensive API reference

2.1.1 Corrfunc package

Corrfunc is a set of high-performance routines for computing clustering statistics on a distribution of points.

Corrfunc.**read_text_file**(*filename*, *encoding=u'utf-8'*)
Reads a file under python3 with encoding (default UTF-8). Also works under python2, without encoding. Uses the EAFP (https://docs.python.org/2/glossary.html#term-eafp) principle.

Corrfunc.**which**(*program*, *mode=1*, *path=None*)
Mimics the Unix utility which. For python3.3+, shutil.which provides all of the required functionality. An implementation is provided in case shutil.which does not exist.

Parameters

- **program** – (required) string Name of program (can be fully-qualified path as well)
- **mode** – (optional) integer flag bits Permissions to check for in the executable Default: os.F_OK (file exists) | os.X_OK (executable file)
- **path** – (optional) string A custom path list to check against. Implementation taken from shutil.py.

Returns A fully qualified path to program as resolved by path or user environment. Returns None when program can not be resolved.

Corrfunc.**write_text_file**(*filename*, *contents*, *encoding=u'utf-8'*)
Writes a file under python3 with encoding (default UTF-8). Also works under python2, without encoding. Uses the EAFP (https://docs.python.org/2/glossary.html#term-eafp) principle.

Subpackages

Corrfunc.mocks package

Wrapper for all clustering statistic calculations on galaxies in a mock catalog.

Corrfunc.mocks.**DDrppi_mocks**(*autocorr*, *cosmology*, *nthreads*, *pimax*, *binfile*, *RA1*, *DEC1*, *CZ1*, *weights1=None*, *RA2=None*, *DEC2=None*, *CZ2=None*, *weights2=None*, *is_comoving_dist=False*, *verbose=False*, *output_rpavg=False*, *fast_divide_and_NR_steps=0*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*, *weight_type=None*)

Calculate the 2-D pair-counts corresponding to the projected correlation function, $\xi(r_p, \pi)$. Pairs which are separated by less than the `rp` bins (specified in `binfile`) in the X-Y plane, and less than `pimax` in the Z-dimension are counted. The input positions are expected to be on-sky co-ordinates. This module is suitable for calculating correlation functions for mock catalogs.

If `weights` are provided, the resulting pair counts are weighted. The weighting scheme depends on `weight_type`.

Returns a numpy structured array containing the pair counts for the specified bins.

Note: that this module only returns pair counts and not the actual correlation function $\xi(r_p, \pi)$ or $wp(r_p)$. See the utilities *Corrfunc.utils.convert_3d_counts_to_cf* and *Corrfunc.utils.convert_rp_pi_counts_to_wp* for computing $\xi(r_p, \pi)$ and $wp(r_p)$ respectively from the pair counts.

Parameters

- **autocorr** (*boolean, required*) – Boolean flag for auto/cross-correlation. If autocorr is set to 1, then the second set of particle positions are not required.
- **cosmology** (*integer, required*) – Integer choice for setting cosmology. Valid values are 1->LasDamas cosmology and 2->Planck cosmology. If you need arbitrary cosmology, easiest way is to convert the `CZ` values into co-moving distance, based on your preferred cosmology. Set `is_comoving_dist=True`, to indicate that the co-moving distance conversion has already been done.

  Choices:

  1. LasDamas cosmology. $\Omega_m = 0.25, \Omega_\Lambda = 0.75$
  2. Planck cosmology. $\Omega_m = 0.302, \Omega_\Lambda = 0.698$

  To setup a new cosmology, add an entry to the function, `init_cosmology` in `ROOT/utils/cosmology_params.c` and re-install the entire package.
- **nthreads** (*integer*) – The number of OpenMP threads to use. Has no effect if OpenMP was not enabled during library compilation.
- **pimax** (*double*) – A double-precision value for the maximum separation along the Z-dimension.

  Distances along the π direction are binned with unit depth. For instance, if `pimax=40`, then 40 bins will be created along the `pi` direction. Only pairs with `0 <= dz < pimax` are counted (no equality).
- **binfile** (*string or an list/array of floats*) – For string input: filename specifying the `rp` bins for `DDrppi_mocks`. The file should contain white-space separated

values of (rpmin, rpmax) for each `rp` wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

For array-like input: A sequence of `rp` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.

- **RA1** (*array-like, real (float/double)*) – The array of Right Ascensions for the first set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

  Calculations are done in the precision of the supplied arrays.

- **DEC1** (*array-like, real (float/double)*) – Array of Declinations for the first set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0].

  Must be of same precision type as RA1.

- **CZ1** (*array-like, real (float/double)*) – Array of (Speed Of Light * Redshift) values for the first set of points. Code will try to detect cases where `redshifts` have been passed and multiply the entire array with the `speed of light`.

  If is_comoving_dist is set, then `CZ1` is interpreted as the co-moving distance, rather than *cz*.

- **weights1** (*array_like, real (float/double), optional*) – A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field. If only one of weights1 and weights2 is specified, the other will be set to uniform weights.

- **RA2** (*array-like, real (float/double)*) – The array of Right Ascensions for the second set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

  Must be of same precision type as RA1/DEC1/CZ1.

- **DEC2** (*array-like, real (float/double)*) – Array of Declinations for the second set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0].

  Must be of same precision type as RA1/DEC1/CZ1.

- **CZ2** (*array-like, real (float/double)*) – Array of (Speed Of Light * Redshift) values for the second set of points. Code will try to detect cases where `redshifts` have been passed and multiply the entire array with the `speed of light`.

  If is_comoving_dist is set, then `CZ2` is interpreted as the co-moving distance, rather than *cz*.

  Must be of same precision type as RA1/DEC1/CZ1.

- **weights2** (*array-like, real (float/double), optional*) – Same as weights1, but for the second set of positions

- **is_comoving_dist** (*boolean (default false)*) – Boolean flag to indicate that `cz` values have already been converted into co-moving distances. This flag allows arbitrary cosmologies to be used in `Corrfunc`.

- **verbose** (*boolean (default false)*) – Boolean flag to control output of informational messages

- **output_rpavg** (*boolean (default false)*) – Boolean flag to output the average `rp` for each bin. Code will run slower if you set this flag.

  If you are calculating in single-precision, `rpavg` will suffer suffer from numerical loss of precision and can not be trusted. If you need accurate `rpavg` values, then pass in double precision arrays for the particle positions.

- **fast_divide_and_NR_steps** (*integer (default 0)*) – Replaces the division in `AVX` implementation with an approximate reciprocal, followed by `fast_divide_and_NR_steps` of Newton-Raphson. Can improve runtime by ~15-20% on older computers. Value of 0 uses the standard division operation.

- **(xyz)bin_refine_factor** (*integer, default is (2,2,1); typically within [1-3]*) – Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

- **max_cells_per_dim** (*integer, default is 100, typical values in [50-300]*) – Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rpmax` is too small relative to the boxsize (and increasing helps the runtime).

- **c_api_timer** (*boolean (default false)*) – Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

- **isa** (string (default `fastest`)) – Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

  Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

  Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

- **weight_type** (*string, optional*) – The type of weighting to apply. One of ["pair_product", None]. Default: None.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [rpmin, rpmax, rpavg, pimax, npairs, weightavg] for each radial bin specified in the `binfile`. If `output_ravg` is not set, then `rpavg` will be set to 0.0 for all bins; similarly for `weightavg`. `npairs` contains the number of pairs in that bin and can be used to compute the actual $\xi(r_p, \pi)$ or $wp(rp)$ by combining with (DR, RR) counts.

- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.mocks.DDrppi_mocks import DDrppi_mocks
>>> import math
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
```

(continues on next page)

(continued from previous page)

```
...                 "../mocks/tests/", "bins")
>>> N = 100000
>>> boxsize = 420.0
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(-0.5*boxsize, 0.5*boxsize, N)
>>> Y = np.random.uniform(-0.5*boxsize, 0.5*boxsize, N)
>>> Z = np.random.uniform(-0.5*boxsize, 0.5*boxsize, N)
>>> weights = np.ones_like(X)
>>> CZ = np.sqrt(X*X + Y*Y + Z*Z)
>>> inv_cz = 1.0/CZ
>>> X *= inv_cz
>>> Y *= inv_cz
>>> Z *= inv_cz
>>> DEC = 90.0 - np.arccos(Z)*180.0/math.pi
>>> RA = (np.arctan2(Y, X)*180.0/math.pi) + 180.0
>>> autocorr = 1
>>> cosmology = 1
>>> nthreads = 2
>>> pimax = 40.0
>>> results = DDrppi_mocks(autocorr, cosmology, nthreads,
...                        pimax, binfile, RA, DEC, CZ,
...                        weights1=weights, weight_type='pair_product',
...                        output_rpavg=True, is_comoving_dist=True)
>>> for r in results[519:]: print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10.1f}"
...                               " {4:10d} {5:10.6f}".format(r['rmin'], r['rmax
↪'],
...                               r['rpavg'], r['pimax'], r['npairs'], r[
↪'weightavg']))
...
 11.359969  16.852277  14.285169       40.0     104850   1.000000
 16.852277  25.000000  21.181246        1.0     274144   1.000000
 16.852277  25.000000  21.190844        2.0     272876   1.000000
 16.852277  25.000000  21.183321        3.0     272294   1.000000
 16.852277  25.000000  21.188486        4.0     272506   1.000000
 16.852277  25.000000  21.170832        5.0     272100   1.000000
 16.852277  25.000000  21.165379        6.0     271788   1.000000
 16.852277  25.000000  21.175246        7.0     270040   1.000000
 16.852277  25.000000  21.187417        8.0     269492   1.000000
 16.852277  25.000000  21.172066        9.0     269682   1.000000
 16.852277  25.000000  21.182460       10.0     268266   1.000000
 16.852277  25.000000  21.170594       11.0     268744   1.000000
 16.852277  25.000000  21.178608       12.0     266820   1.000000
 16.852277  25.000000  21.187184       13.0     266510   1.000000
 16.852277  25.000000  21.184937       14.0     265484   1.000000
 16.852277  25.000000  21.180184       15.0     265258   1.000000
 16.852277  25.000000  21.191504       16.0     262952   1.000000
 16.852277  25.000000  21.187746       17.0     262602   1.000000
 16.852277  25.000000  21.189778       18.0     260206   1.000000
 16.852277  25.000000  21.188882       19.0     259410   1.000000
 16.852277  25.000000  21.185684       20.0     256806   1.000000
 16.852277  25.000000  21.194036       21.0     255574   1.000000
 16.852277  25.000000  21.184115       22.0     255406   1.000000
 16.852277  25.000000  21.178255       23.0     252394   1.000000
 16.852277  25.000000  21.184644       24.0     252220   1.000000
 16.852277  25.000000  21.187020       25.0     251668   1.000000
 16.852277  25.000000  21.183827       26.0     249648   1.000000
```

(continues on next page)

(continued from previous page)

```
16.852277  25.000000  21.183121       27.0     247160   1.000000
16.852277  25.000000  21.180872       28.0     246238   1.000000
16.852277  25.000000  21.185251       29.0     246030   1.000000
16.852277  25.000000  21.183488       30.0     242124   1.000000
16.852277  25.000000  21.194538       31.0     242426   1.000000
16.852277  25.000000  21.190702       32.0     239778   1.000000
16.852277  25.000000  21.188985       33.0     239046   1.000000
16.852277  25.000000  21.187092       34.0     237640   1.000000
16.852277  25.000000  21.185515       35.0     236256   1.000000
16.852277  25.000000  21.190278       36.0     233536   1.000000
16.852277  25.000000  21.183240       37.0     233274   1.000000
16.852277  25.000000  21.183796       38.0     231628   1.000000
16.852277  25.000000  21.200668       39.0     230378   1.000000
16.852277  25.000000  21.181153       40.0     229006   1.000000
```

Corrfunc.mocks.**DDtheta_mocks**(*autocorr*, *nthreads*, *binfile*, *RA1*, *DEC1*, *weights1=None*, *RA2=None*, *DEC2=None*, *weights2=None*, *link_in_dec=True*, *link_in_ra=True*, *verbose=False*, *output_thetaavg=False*, *fast_acos=False*, *ra_refine_factor=2*, *dec_refine_factor=2*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*, *weight_type=None*)

Function to compute the angular correlation function for points on the sky (i.e., mock catalogs or observed galaxies).

Returns a numpy structured array containing the pair counts for the specified angular bins.

If `weights` are provided, the resulting pair counts are weighted. The weighting scheme depends on `weight_type`.

Note: This module only returns pair counts and not the actual correlation function $\omega(heta)$. See *Corrfunc.utils.convert_3d_counts_to_cf* for computing $\omega(heta)$ from the pair counts returned.

Parameters

- **autocorr** (*boolean, required*) – Boolean flag for auto/cross-correlation. If autocorr is set to 1, then the second set of particle positions are not required.
- **nthreads** (*integer*) – Number of threads to use.
- **binfile** (*string or an list/array of floats. Units: degrees.*) – For string input: filename specifying the `theta` bins for `DDtheta_mocks`. The file should contain white-space separated values of (thetamin, thetamax) for each `theta` wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

  For array-like input: A sequence of `theta` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0 degrees. This array does not need to be sorted.
- **RA1** (*array-like, real (float/double)*) – The array of Right Ascensions for the first set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

  Calculations are done in the precision of the supplied arrays.

- **DEC1** (*array-like, real (float/double)*) – Array of Declinations for the first set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0]. Must be of same precision type as RA1.
- **weights1** (*array_like, real (float/double), optional*) – A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field. If only one of weights1 and weights2 is specified, the other will be set to uniform weights.
- **RA2** (*array-like, real (float/double)*) – The array of Right Ascensions for the second set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0]. Must be of same precision type as RA1/DEC1.
- **DEC2** (*array-like, real (float/double)*) – Array of Declinations for the second set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0]. Must be of same precision type as RA1/DEC1.
- **weights2** (*array-like, real (float/double), optional*) – Same as weights1, but for the second set of positions
- **link_in_dec** (*boolean (default True)*) – Boolean flag to create lattice in Declination. Code runs faster with this option. However, if the angular separations are too small, then linking in declination might produce incorrect results. When running for the first time, check your results by comparing with the output of the code for `link_in_dec=False` and `link_in_ra=False`.
- **link_in_ra** (*boolean (default True)*) – Boolean flag to create lattice in Right Ascension. Setting this option implies `link_in_dec=True`. Similar considerations as `link_in_dec` described above.

  If you disable both `link_in_dec` and `link_in_ra`, then the code reduces to a brute-force pair counter. No lattices are created at all. For very small angular separations, the brute-force method might be the most numerically stable method.
- **verbose** (*boolean (default false)*) – Boolean flag to control output of informational messages
- **output_thetaavg** (*boolean (default false)*) – Boolean flag to output the average '' heta'' for each bin. Code will run slower if you set this flag.

  If you are calculating in single-precision, `thetaavg` will suffer from numerical loss of precision and can not be trusted. If you need accurate `thetaavg` values, then pass in double precision arrays for `RA/DEC`.

  Code will run significantly slower if you enable this option. Use the keyword `fast_acos` if you can tolerate some loss of precision.
- **fast_acos** (*boolean (default false)*) – Flag to use numerical approximation for the `arccos` - gives better performance at the expense of some precision. Relevant only if `output_thetaavg==True`.

  Developers: Two versions already coded up in `utils/fast_acos.h`, so you can choose the version you want. There are also notes on how to implement faster (and less accurate) functions, particularly relevant if you know your `theta` range is limited. If you implement a new version, then you will have to reinstall the entire Corrfunc package.

  Note: Tests will fail if you run the tests with''fast_acos=True''.

- **(radec)_refine_factor** (*integer, default is (2,2); typically within [1-3]*) – Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

  Only two refine factors are to be specified and these correspond to `ra` and `dec` (rather, than the usual three of `(xyz)bin_refine_factor` for all other correlation functions).

- **max_cells_per_dim** (*integer, default is 100, typical values in [50-300]*) – Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `thetamax` is too small relative to the boxsize (and increasing helps the runtime).

- **c_api_timer** (*boolean (default false)*) – Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

- **isa** (string (default `fastest`)) – Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

  Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

  Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [thetamin, thetamax, thetaavg, npairs, weightavg] for each angular bin specified in the `binfile`. If `output_thetaavg` is not set then `thetavg` will be set to 0.0 for all bins; similarly for `weightavg`. `npairs` contains the number of pairs in that bin.

- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> import time
>>> from math import pi
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.mocks.DDtheta_mocks import DDtheta_mocks
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../mocks/tests/", "angular_bins")
>>> N = 100000
>>> nthreads = 4
>>> seed = 42
>>> np.random.seed(seed)
>>> RA = np.random.uniform(0.0, 2.0*pi, N)*180.0/pi
>>> cos_theta = np.random.uniform(-1.0, 1.0, N)
>>> DEC = 90.0 - np.arccos(cos_theta)*180.0/pi
>>> weights = np.ones_like(RA)
>>> autocorr = 1
>>> for isa in ['AVX', 'SSE42', 'FALLBACK']:
...     for link_in_dec in [False, True]:
```

(continues on next page)

(continued from previous page)

```
...         for link_in_ra in [False, True]:
...             results = DDtheta_mocks(autocorr, nthreads, binfile,
...                         RA, DEC, output_thetaavg=True,
...                         weights1=weights, weight_type='pair_product',
...                         link_in_dec=link_in_dec, link_in_ra=link_in_ra,
...                         isa=isa, verbose=True)
>>> for r in results: print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10d} {4:10.6f}".
...                         format(r['thetamin'], r['thetamax'],
...                         r['thetaavg'], r['npairs'], r['weightavg']))
...
  0.010000   0.014125   0.012272         62   1.000000
  0.014125   0.019953   0.016978        172   1.000000
  0.019953   0.028184   0.024380        298   1.000000
  0.028184   0.039811   0.034321        598   1.000000
  0.039811   0.056234   0.048535       1164   1.000000
  0.056234   0.079433   0.068385       2438   1.000000
  0.079433   0.112202   0.096631       4658   1.000000
  0.112202   0.158489   0.136834       9414   1.000000
  0.158489   0.223872   0.192967      19098   1.000000
  0.223872   0.316228   0.272673      37848   1.000000
  0.316228   0.446684   0.385344      75520   1.000000
  0.446684   0.630957   0.543973     150938   1.000000
  0.630957   0.891251   0.768406     301854   1.000000
  0.891251   1.258925   1.085273     599896   1.000000
  1.258925   1.778279   1.533461    1200238   1.000000
  1.778279   2.511886   2.166009    2396338   1.000000
  2.511886   3.548134   3.059159    4775162   1.000000
  3.548134   5.011872   4.321445    9532582   1.000000
  5.011872   7.079458   6.104214   19001930   1.000000
  7.079458  10.000000   8.622400   37842502   1.000000
```

Corrfunc.mocks.**vpf_mocks**(*rmax*, *nbins*, *nspheres*, *numpN*, *threshold_ngb*, *centers_file*, *cosmology*, *RA*, *DEC*, *CZ*, *RAND_RA*, *RAND_DEC*, *RAND_CZ*, *verbose=False*, *is_comoving_dist=False*, *xbin_refine_factor=1*, *ybin_refine_factor=1*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*)

Function to compute the counts-in-cells on points on the sky. Suitable for mock catalogs and observed galaxies.

Returns a numpy structured array containing the probability of a sphere of radius up to `rmax` containing `0--numpN-1` galaxies.

Parameters

- **rmax** (*double*) – Maximum radius of the sphere to place on the particles
- **nbins** (*integer*) – Number of bins in the counts-in-cells. Radius of first shell is rmax/nbins
- **nspheres** (*integer (>= 0)*) – Number of random spheres to place within the particle distribution. For a small number of spheres, the error is larger in the measured pN's.
- **numpN** (*integer (>= 1)*) – Governs how many unique pN's are to returned. If `numpN` is set to 1, then only the vpf (p0) is returned. For `numpN=2`, p0 and p1 are returned.

  More explicitly, the columns in the results look like the following:

numpN	Columns in output
1	p0
2	p0 p1
3	p0 p1 p2
4	p0 p1 p2 p3

and so on...

Note: `p0` is the vpf

- **`threshold_ngb`** (*`integer`*) – Minimum number of random points needed in a `rmax` sphere such that it is considered to be entirely within the mock footprint. The command-line version, `mocks/vpf/vpf_mocks.c`, assumes that the minimum number of randoms can be at most a 1-sigma deviation from the expected random number density.

- **`centers_file`** (*`string, filename`*) – A file containing random sphere centers. If the file does not exist, then a list of random centers will be written out. In that case, the randoms arrays, `RAND_RA`, `RAND_DEC` and `RAND_CZ` are used to check that the sphere is entirely within the footprint. If the file does exist but either `rmax` is too small or there are not enough centers then the file will be overwritten.

  Note: If the centers file has to be written, the code will take significantly longer to finish. However, subsequent runs can re-use that centers file and will be faster.

- **`cosmology`** (*`integer, required`*) – Integer choice for setting cosmology. Valid values are 1->LasDamas cosmology and 2->Planck cosmology. If you need arbitrary cosmology, easiest way is to convert the `CZ` values into co-moving distance, based on your preferred cosmology. Set `is_comoving_dist=True`, to indicate that the co-moving distance conversion has already been done.

  Choices:

  1. LasDamas cosmology. $\Omega_m = 0.25$, $\Omega_\Lambda = 0.75$
  2. Planck cosmology. $\Omega_m = 0.302$, $\Omega_\Lambda = 0.698$

  To setup a new cosmology, add an entry to the function, `init_cosmology` in `ROOT/utils/cosmology_params.c` and re-install the entire package.

- **`RA`** (*`array-like, real (float/double)`*) – The array of Right Ascensions for the first set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

  Calculations are done in the precision of the supplied arrays.

- **`DEC`** (*`array-like, real (float/double)`*) – Array of Declinations for the first set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0].

  Must be of same precision type as RA.

- **`CZ`** (*`array-like, real (float/double)`*) – Array of (Speed Of Light * Redshift) values for the first set of points. Code will try to detect cases where `redshifts` have been passed and multiply the entire array with the `speed of light`.

  If `is_comoving_dist` is set, then `CZ` is interpreted as the co-moving distance, rather than (Speed Of Light * Redshift).

- **`RAND_RA`** (*`array-like, real (float/double)`*) – The array of Right Ascensions for the randoms. RA's are expected to be in [0.0, 360.0], but the code will try to

fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

Must be of same precision type as RA/DEC/CZ.

- **RAND_DEC** (*array-like, real (float/double)*) – Array of Declinations for the randoms. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0].

  Must be of same precision type as RA/DEC/CZ.

- **RAND_CZ** (*array-like, real (float/double)*) – Array of (Speed Of Light * Redshift) values for the randoms. Code will try to detect cases where `redshifts` have been passed and multiply the entire array with the `speed of light`.

  If `is_comoving_dist` is set, then `CZ2` is interpreted as the co-moving distance, rather than `(Speed Of Light * Redshift)`.

  Note: RAND_RA, RAND_DEC and RAND_CZ are only used when the `centers_file` needs to be written out. In that case, the RAND_RA, RAND_DEC, and RAND_CZ are used as random centers.

- **verbose** (*boolean (default false)*) – Boolean flag to control output of informational messages

- **is_comoving_dist** (*boolean (default false)*) – Boolean flag to indicate that `cz` values have already been converted into co-moving distances. This flag allows arbitrary cosmologies to be used in `Corrfunc`.

- **(xyz)bin_refine_factor** (*integer, default is (1,1,1); typically within [1-3]*) – Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

  Note: Since the counts in spheres calculation is symmetric in all 3 dimensions, the defaults are different from the clustering routines.

- **max_cells_per_dim** (*integer, default is 100, typical values in [50-300]*) – Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rmax` is too small relative to the boxsize (and increasing helps the runtime).

- **c_api_timer** (*boolean (default false)*) – Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

- **isa** (string (default `fastest`)) – Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

  Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

  Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [rmax, pN[numpN]] with `nbins` elements. Each row contains the maximum radius of the sphere and the `numpN` elements in the `pN` array. Each element of this array contains the probability

that a sphere of radius `rmax` contains *exactly* `N` galaxies. For example, pN[0] (p0, the void probibility function) is the probability that a sphere of radius `rmax` contains 0 galaxies.

- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Example

```
>>> from __future__ import print_function
>>> import math
>>> from os.path import dirname, abspath, join as pjoin
>>> import numpy as np
>>> import Corrfunc
>>> from Corrfunc.mocks.vpf_mocks import vpf_mocks
>>> rmax = 10.0
>>> nbins = 10
>>> numbins_to_print = nbins
>>> nspheres = 10000
>>> numpN = 6
>>> threshold_ngb = 1  # does not matter since we have the centers
>>> cosmology = 1  # LasDamas cosmology
>>> centers_file = pjoin(dirname(abspath(Corrfunc.__file__)),
...                      "../mocks/tests/data/",
...                      "Mr19_centers_xyz_forVPF_rmax_10Mpc.txt")
>>> N = 1000000
>>> boxsize = 420.0
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(-0.5*boxsize, 0.5*boxsize, N)
>>> Y = np.random.uniform(-0.5*boxsize, 0.5*boxsize, N)
>>> Z = np.random.uniform(-0.5*boxsize, 0.5*boxsize, N)
>>> CZ = np.sqrt(X*X + Y*Y + Z*Z)
>>> inv_cz = 1.0/CZ
>>> X *= inv_cz
>>> Y *= inv_cz
>>> Z *= inv_cz
>>> DEC = 90.0 - np.arccos(Z)*180.0/math.pi
>>> RA = (np.arctan2(Y, X)*180.0/math.pi) + 180.0
>>> results = vpf_mocks(rmax, nbins, nspheres, numpN, threshold_ngb,
...                     centers_file, cosmology,
...                     RA, DEC, CZ,
...                     RA, DEC, CZ,
...                     is_comoving_dist=True)
>>> for r in results:
...     print("{0:10.1f} ".format(r[0]), end="")
...
...     for pn in r[1]:
...         print("{0:10.3f} ".format(pn), end="")
...
...     print("")
   1.0      0.999      0.001      0.000      0.000      0.000      0.000
   2.0      0.992      0.007      0.001      0.000      0.000      0.000
   3.0      0.982      0.009      0.005      0.002      0.001      0.000
   4.0      0.975      0.006      0.006      0.005      0.003      0.003
   5.0      0.971      0.004      0.003      0.003      0.004      0.003
   6.0      0.967      0.003      0.003      0.001      0.003      0.002
```

(continues on next page)

(continued from previous page)

7.0	0.962	0.004	0.002	0.003	0.002	0.001
8.0	0.958	0.004	0.002	0.003	0.001	0.002
9.0	0.953	0.003	0.003	0.002	0.003	0.001
10.0	0.950	0.003	0.002	0.002	0.001	0.002

Corrfunc.mocks.**DDsmu_mocks**(*autocorr*, *cosmology*, *nthreads*, *mu_max*, *nmu_bins*, *binfile*, *RA1*, *DEC1*, *CZ1*, *weights1=None*, *RA2=None*, *DEC2=None*, *CZ2=None*, *weights2=None*, *is_comoving_dist=False*, *verbose=False*, *output_savg=False*, *fast_divide_and_NR_steps=0*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*, *weight_type=None*)

Calculate the 2-D pair-counts corresponding to the projected correlation function, $\xi(s, \mu)$. The pairs are counted in bins of radial separation and cosine of angle to the line-of-sight (LOS). The input positions are expected to be on-sky co-ordinates. This module is suitable for calculating correlation functions for mock catalogs.

If `weights` are provided, the resulting pair counts are weighted. The weighting scheme depends on `weight_type`.

Returns a numpy structured array containing the pair counts for the specified bins.

Note: This module only returns pair counts and not the actual correlation function $\xi(s, \mu)$. See the utilities *Corrfunc.utils.convert_3d_counts_to_cf* for computing $\xi(s, \mu)$ from the pair counts.

New in version 2.1.0.

Parameters

- **autocorr** (*boolean, required*) – Boolean flag for auto/cross-correlation. If autocorr is set to 1, then the second set of particle positions are not required.
- **cosmology** (*integer, required*) – Integer choice for setting cosmology. Valid values are 1->LasDamas cosmology and 2->Planck cosmology. If you need arbitrary cosmology, easiest way is to convert the `CZ` values into co-moving distance, based on your preferred cosmology. Set `is_comoving_dist=True`, to indicate that the co-moving distance conversion has already been done.

  Choices:

  1. LasDamas cosmology. $\Omega_m = 0.25$, $\Omega_\Lambda = 0.75$
  2. Planck cosmology. $\Omega_m = 0.302$, $\Omega_\Lambda = 0.698$

  To setup a new cosmology, add an entry to the function, `init_cosmology` in `ROOT/utils/cosmology_params.c` and re-install the entire package.
- **nthreads** (*integer*) – The number of OpenMP threads to use. Has no effect if OpenMP was not enabled during library compilation.
- **mu_max** (*double. Must be in range [0.0, 1.0]*) – A double-precision value for the maximum cosine of the angular separation from the line of sight (LOS). Here, `mu` is defined as the angle between `s` and `l`. If v_1 and v_2 represent the vectors to each point constituting the pair, then $s := v_1 - v_2$ and $l := 1/2(v_1 + v_2)$.

  Note: Only pairs with $0 <= \cos(\theta_{LOS}) < \mu_{max}$ are counted (no equality).
- **nmu_bins** (*int*) – The number of linear `mu` bins, with the bins ranging from from (0, μ_{max})

- **binfile** (*string or an list/array of floats*) – For string input: filename specifying the `s` bins for `DDsmu_mocks`. The file should contain white-space separated values of (smin, smax) specifying each `s` bin wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

  For array-like input: A sequence of `s` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.

- **RA1** (*array-like, real (float/double)*) – The array of Right Ascensions for the first set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

  Calculations are done in the precision of the supplied arrays.

- **DEC1** (*array-like, real (float/double)*) – Array of Declinations for the first set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0].

  Must be of same precision type as RA1.

- **CZ1** (*array-like, real (float/double)*) – Array of (Speed Of Light * Redshift) values for the first set of points. Code will try to detect cases where `redshifts` have been passed and multiply the entire array with the `speed of light`.

  If is_comoving_dist is set, then `CZ1` is interpreted as the co-moving distance, rather than *cz*.

- **weights1** (*array_like, real (float/double), optional*) – A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field. If only one of `weights1` or `weights2` is specified, the other will be set to uniform weights.

- **RA2** (*array-like, real (float/double)*) – The array of Right Ascensions for the second set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

  Must be of same precision type as RA1/DEC1/CZ1.

- **DEC2** (*array-like, real (float/double)*) – Array of Declinations for the second set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0].

  Must be of same precision type as RA1/DEC1/CZ1.

- **CZ2** (*array-like, real (float/double)*) – Array of (Speed Of Light * Redshift) values for the second set of points. Code will try to detect cases where `redshifts` have been passed and multiply the entire array with the `speed of light`.

  If is_comoving_dist is set, then `CZ2` is interpreted as the co-moving distance, rather than *cz*.

  Must be of same precision type as RA1/DEC1/CZ1.

- **weights2** (*array-like, real (float/double), optional*) – Same as weights1, but for the second set of positions

- **is_comoving_dist** (*boolean (default false)*) – Boolean flag to indicate that `cz` values have already been converted into co-moving distances. This flag allows arbitrary cosmologies to be used in `Corrfunc`.

- **verbose** (*boolean (default false)*) – Boolean flag to control output of informational messages
- **output_savg** (*boolean (default false)*) – Boolean flag to output the average `s` for each bin. Code will run slower if you set this flag. Also, note, if you are calculating in single-precision, `savg` will suffer from numerical loss of precision and can not be trusted. If you need accurate `savg` values, then pass in double precision arrays for the particle positions.
- **fast_divide_and_NR_steps** (*integer (default 0)*) – Replaces the division in `AVX` implementation with an approximate reciprocal, followed by `fast_divide_and_NR_steps` of Newton-Raphson. Can improve runtime by ~15-20% on older computers. Value of 0 uses the standard division operation.
- **(xyz)bin_refine_factor** (*integer, default is (2,2,1); typically within [1-3]*) – Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.
- **max_cells_per_dim** (*integer, default is 100, typical values in [50-300]*) – Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rpmax` is too small relative to the boxsize (and increasing helps the runtime).
- **c_api_timer** (*boolean (default false)*) – Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.
- **isa** (string (default `fastest`)) – Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

  Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

  Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.
- **weight_type** (*string, optional*) – The type of weighting to apply. One of ["pair_product", None]. Default: None.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [smin, smax, savg, mumax, npairs, weightavg] for each separation bin specified in the `binfile`. If `output_savg` is not set, then `savg` will be set to 0.0 for all bins; similarly for `weightavg`. `npairs` contains the number of pairs in that bin and can be used to compute the actual $\xi(s, \mu)$ by combining with (DR, RR) counts.
- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Submodules

Corrfunc.mocks.DDrppi_mocks module

Python wrapper around the C extension for the pair counter in `mocks/DDrppi_mocks/`. This python wrapper is *Corrfunc.mocks.DDrppi_mocks*

Corrfunc.mocks.DDrppi_mocks.**DDrppi_mocks**(*autocorr*, *cosmology*, *nthreads*, *pimax*, *binfile*, *RA1*, *DEC1*, *CZ1*, *weights1=None*, *RA2=None*, *DEC2=None*, *CZ2=None*, *weights2=None*, *is_comoving_dist=False*, *verbose=False*, *output_rpavg=False*, *fast_divide_and_NR_steps=0*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*, *weight_type=None*)

Calculate the 2-D pair-counts corresponding to the projected correlation function, $\xi(r_p, \pi)$. Pairs which are separated by less than the `rp` bins (specified in `binfile`) in the X-Y plane, and less than `pimax` in the Z-dimension are counted. The input positions are expected to be on-sky co-ordinates. This module is suitable for calculating correlation functions for mock catalogs.

If `weights` are provided, the resulting pair counts are weighted. The weighting scheme depends on `weight_type`.

Returns a numpy structured array containing the pair counts for the specified bins.

Note: that this module only returns pair counts and not the actual correlation function $\xi(r_p, \pi)$ or $wp(r_p)$. See the utilities *Corrfunc.utils.convert_3d_counts_to_cf* and *Corrfunc.utils.convert_rp_pi_counts_to_wp* for computing $\xi(r_p, \pi)$ and $wp(r_p)$ respectively from the pair counts.

Parameters

- **autocorr** (*boolean, required*) – Boolean flag for auto/cross-correlation. If autocorr is set to 1, then the second set of particle positions are not required.
- **cosmology** (*integer, required*) – Integer choice for setting cosmology. Valid values are 1->LasDamas cosmology and 2->Planck cosmology. If you need arbitrary cosmology, easiest way is to convert the `CZ` values into co-moving distance, based on your preferred cosmology. Set `is_comoving_dist=True`, to indicate that the co-moving distance conversion has already been done.

  Choices:

  1. LasDamas cosmology. $\Omega_m = 0.25$, $\Omega_\Lambda = 0.75$
  2. Planck cosmology. $\Omega_m = 0.302$, $\Omega_\Lambda = 0.698$

  To setup a new cosmology, add an entry to the function, `init_cosmology` in `ROOT/utils/cosmology_params.c` and re-install the entire package.
- **nthreads** (*integer*) – The number of OpenMP threads to use. Has no effect if OpenMP was not enabled during library compilation.
- **pimax** (*double*) – A double-precision value for the maximum separation along the Z-dimension.

  Distances along the π direction are binned with unit depth. For instance, if `pimax=40`, then 40 bins will be created along the `pi` direction. Only pairs with `0 <= dz < pimax` are counted (no equality).
- **binfile** (*string or an list/array of floats*) – For string input: filename specifying the `rp` bins for `DDrppi_mocks`. The file should contain white-space separated values of (rpmin, rpmax) for each `rp` wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

For array-like input: A sequence of `rp` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.

- **RA1** (*array-like, real (float/double)*) – The array of Right Ascensions for the first set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

  Calculations are done in the precision of the supplied arrays.

- **DEC1** (*array-like, real (float/double)*) – Array of Declinations for the first set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0].

  Must be of same precision type as RA1.

- **CZ1** (*array-like, real (float/double)*) – Array of (Speed Of Light * Redshift) values for the first set of points. Code will try to detect cases where `redshifts` have been passed and multiply the entire array with the `speed of light`.

  If is_comoving_dist is set, then `CZ1` is interpreted as the co-moving distance, rather than *cz*.

- **weights1** (*array_like, real (float/double), optional*) – A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field. If only one of weights1 and weights2 is specified, the other will be set to uniform weights.

- **RA2** (*array-like, real (float/double)*) – The array of Right Ascensions for the second set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

  Must be of same precision type as RA1/DEC1/CZ1.

- **DEC2** (*array-like, real (float/double)*) – Array of Declinations for the second set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0].

  Must be of same precision type as RA1/DEC1/CZ1.

- **CZ2** (*array-like, real (float/double)*) – Array of (Speed Of Light * Redshift) values for the second set of points. Code will try to detect cases where `redshifts` have been passed and multiply the entire array with the `speed of light`.

  If is_comoving_dist is set, then `CZ2` is interpreted as the co-moving distance, rather than *cz*.

  Must be of same precision type as RA1/DEC1/CZ1.

- **weights2** (*array-like, real (float/double), optional*) – Same as weights1, but for the second set of positions

- **is_comoving_dist** (*boolean (default false)*) – Boolean flag to indicate that `cz` values have already been converted into co-moving distances. This flag allows arbitrary cosmologies to be used in `Corrfunc`.

- **verbose** (*boolean (default false)*) – Boolean flag to control output of informational messages

- **output_rpavg** (*boolean (default false)*) – Boolean flag to output the average `rp` for each bin. Code will run slower if you set this flag.

If you are calculating in single-precision, `rpavg` will suffer suffer from numerical loss of precision and can not be trusted. If you need accurate `rpavg` values, then pass in double precision arrays for the particle positions.

- **`fast_divide_and_NR_steps`** (*integer (default 0)*) – Replaces the division in `AVX` implementation with an approximate reciprocal, followed by `fast_divide_and_NR_steps` of Newton-Raphson. Can improve runtime by ~15-20% on older computers. Value of 0 uses the standard division operation.
- **`(xyz)bin_refine_factor`** (*integer, default is (2,2,1); typically within [1-3]*) – Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.
- **`max_cells_per_dim`** (*integer, default is 100, typical values in [50-300]*) – Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rpmax` is too small relative to the boxsize (and increasing helps the runtime).
- **`c_api_timer`** (*boolean (default false)*) – Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.
- **`isa`** (string (default `fastest`)) – Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

  Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

  Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.
- **`weight_type`** (*string, optional*) – The type of weighting to apply. One of ["pair_product", None]. Default: None.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [rpmin, rpmax, rpavg, pimax, npairs, weightavg] for each radial bin specified in the `binfile`. If `output_ravg` is not set, then `rpavg` will be set to 0.0 for all bins; similarly for `weightavg`. `npairs` contains the number of pairs in that bin and can be used to compute the actual $\xi(r_p, \pi)$ or $wp(rp)$ by combining with (DR, RR) counts.
- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.mocks.DDrppi_mocks import DDrppi_mocks
>>> import math
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../mocks/tests/", "bins")
>>> N = 100000
>>> boxsize = 420.0
```

(continues on next page)

(continued from previous page)

```
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(-0.5*boxsize, 0.5*boxsize, N)
>>> Y = np.random.uniform(-0.5*boxsize, 0.5*boxsize, N)
>>> Z = np.random.uniform(-0.5*boxsize, 0.5*boxsize, N)
>>> weights = np.ones_like(X)
>>> CZ = np.sqrt(X*X + Y*Y + Z*Z)
>>> inv_cz = 1.0/CZ
>>> X *= inv_cz
>>> Y *= inv_cz
>>> Z *= inv_cz
>>> DEC = 90.0 - np.arccos(Z)*180.0/math.pi
>>> RA = (np.arctan2(Y, X)*180.0/math.pi) + 180.0
>>> autocorr = 1
>>> cosmology = 1
>>> nthreads = 2
>>> pimax = 40.0
>>> results = DDrppi_mocks(autocorr, cosmology, nthreads,
...                        pimax, binfile, RA, DEC, CZ,
...                        weights1=weights, weight_type='pair_product',
...                        output_rpavg=True, is_comoving_dist=True)
>>> for r in results[519:]: print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10.1f}"
...                               " {4:10d} {5:10.6f}".format(r['rmin'], r['rmax
↪'],
...                               r['rpavg'], r['pimax'], r['npairs'], r[
↪'weightavg']))
...
 11.359969  16.852277  14.285169       40.0     104850   1.000000
 16.852277  25.000000  21.181246        1.0     274144   1.000000
 16.852277  25.000000  21.190844        2.0     272876   1.000000
 16.852277  25.000000  21.183321        3.0     272294   1.000000
 16.852277  25.000000  21.188486        4.0     272506   1.000000
 16.852277  25.000000  21.170832        5.0     272100   1.000000
 16.852277  25.000000  21.165379        6.0     271788   1.000000
 16.852277  25.000000  21.175246        7.0     270040   1.000000
 16.852277  25.000000  21.187417        8.0     269492   1.000000
 16.852277  25.000000  21.172066        9.0     269682   1.000000
 16.852277  25.000000  21.182460       10.0     268266   1.000000
 16.852277  25.000000  21.170594       11.0     268744   1.000000
 16.852277  25.000000  21.178608       12.0     266820   1.000000
 16.852277  25.000000  21.187184       13.0     266510   1.000000
 16.852277  25.000000  21.184937       14.0     265484   1.000000
 16.852277  25.000000  21.180184       15.0     265258   1.000000
 16.852277  25.000000  21.191504       16.0     262952   1.000000
 16.852277  25.000000  21.187746       17.0     262602   1.000000
 16.852277  25.000000  21.189778       18.0     260206   1.000000
 16.852277  25.000000  21.188882       19.0     259410   1.000000
 16.852277  25.000000  21.185684       20.0     256806   1.000000
 16.852277  25.000000  21.194036       21.0     255574   1.000000
 16.852277  25.000000  21.184115       22.0     255406   1.000000
 16.852277  25.000000  21.178255       23.0     252394   1.000000
 16.852277  25.000000  21.184644       24.0     252220   1.000000
 16.852277  25.000000  21.187020       25.0     251668   1.000000
 16.852277  25.000000  21.183827       26.0     249648   1.000000
 16.852277  25.000000  21.183121       27.0     247160   1.000000
 16.852277  25.000000  21.180872       28.0     246238   1.000000
 16.852277  25.000000  21.185251       29.0     246030   1.000000
```

(continues on next page)

(continued from previous page)

```
16.852277  25.000000  21.183488       30.0     242124   1.000000
16.852277  25.000000  21.194538       31.0     242426   1.000000
16.852277  25.000000  21.190702       32.0     239778   1.000000
16.852277  25.000000  21.188985       33.0     239046   1.000000
16.852277  25.000000  21.187092       34.0     237640   1.000000
16.852277  25.000000  21.185515       35.0     236256   1.000000
16.852277  25.000000  21.190278       36.0     233536   1.000000
16.852277  25.000000  21.183240       37.0     233274   1.000000
16.852277  25.000000  21.183796       38.0     231628   1.000000
16.852277  25.000000  21.200668       39.0     230378   1.000000
16.852277  25.000000  21.181153       40.0     229006   1.000000
```

Corrfunc.mocks.DDsmu_mocks module

Python wrapper around the C extension for the pair counter in `mocks/DDsmu`. This python wrapper is *Corrfunc.mocks.DDsmu_mocks*

Corrfunc.mocks.DDsmu_mocks.**DDsmu_mocks**(*autocorr*, *cosmology*, *nthreads*, *mu_max*, *nmu_bins*, *binfile*, *RA1*, *DEC1*, *CZ1*, *weights1=None*, *RA2=None*, *DEC2=None*, *CZ2=None*, *weights2=None*, *is_comoving_dist=False*, *verbose=False*, *output_savg=False*, *fast_divide_and_NR_steps=0*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*, *weight_type=None*)

Calculate the 2-D pair-counts corresponding to the projected correlation function, $\xi(s, \mu)$. The pairs are counted in bins of radial separation and cosine of angle to the line-of-sight (LOS). The input positions are expected to be on-sky co-ordinates. This module is suitable for calculating correlation functions for mock catalogs.

If `weights` are provided, the resulting pair counts are weighted. The weighting scheme depends on `weight_type`.

Returns a numpy structured array containing the pair counts for the specified bins.

Note: This module only returns pair counts and not the actual correlation function $\xi(s, \mu)$. See the utilities *Corrfunc.utils.convert_3d_counts_to_cf* for computing $\xi(s, \mu)$ from the pair counts.

New in version 2.1.0.

Parameters

- **autocorr** (*boolean, required*) – Boolean flag for auto/cross-correlation. If autocorr is set to 1, then the second set of particle positions are not required.
- **cosmology** (*integer, required*) – Integer choice for setting cosmology. Valid values are 1->LasDamas cosmology and 2->Planck cosmology. If you need arbitrary cosmology, easiest way is to convert the `CZ` values into co-moving distance, based on your preferred cosmology. Set `is_comoving_dist=True`, to indicate that the co-moving distance conversion has already been done.

  Choices:

  1. LasDamas cosmology. $\Omega_m = 0.25, \Omega_\Lambda = 0.75$
  2. Planck cosmology. $\Omega_m = 0.302, \Omega_\Lambda = 0.698$

To setup a new cosmology, add an entry to the function, `init_cosmology` in `ROOT/utils/cosmology_params.c` and re-install the entire package.

- **nthreads** (*integer*) – The number of OpenMP threads to use. Has no effect if OpenMP was not enabled during library compilation.
- **mu_max** (*double. Must be in range [0.0, 1.0]*) – A double-precision value for the maximum cosine of the angular separation from the line of sight (LOS). Here, `mu` is defined as the angle between `s` and `l`. If v_1 and v_2 represent the vectors to each point constituting the pair, then $s := v_1 - v_2$ and $l := 1/2(v_1 + v_2)$.

  Note: Only pairs with $0 <= \cos(\theta_{LOS}) < \mu_{max}$ are counted (no equality).
- **nmu_bins** (*int*) – The number of linear `mu` bins, with the bins ranging from from (0, μ_{max})
- **binfile** (*string or an list/array of floats*) – For string input: filename specifying the `s` bins for `DDsmu_mocks`. The file should contain white-space separated values of (smin, smax) specifying each `s` bin wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

  For array-like input: A sequence of `s` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.
- **RA1** (*array-like, real (float/double)*) – The array of Right Ascensions for the first set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

  Calculations are done in the precision of the supplied arrays.
- **DEC1** (*array-like, real (float/double)*) – Array of Declinations for the first set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0].

  Must be of same precision type as RA1.
- **CZ1** (*array-like, real (float/double)*) – Array of (Speed Of Light * Redshift) values for the first set of points. Code will try to detect cases where `redshifts` have been passed and multiply the entire array with the `speed of light`.

  If is_comoving_dist is set, then `CZ1` is interpreted as the co-moving distance, rather than *cz*.
- **weights1** (*array_like, real (float/double), optional*) – A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field. If only one of `weights1` or `weights2` is specified, the other will be set to uniform weights.
- **RA2** (*array-like, real (float/double)*) – The array of Right Ascensions for the second set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

  Must be of same precision type as RA1/DEC1/CZ1.
- **DEC2** (*array-like, real (float/double)*) – Array of Declinations for the second set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0].

Must be of same precision type as RA1/DEC1/CZ1.

- **CZ2** (*array-like, real (float/double)*) – Array of (Speed Of Light * Redshift) values for the second set of points. Code will try to detect cases where `redshifts` have been passed and multiply the entire array with the `speed of light`.

  If is_comoving_dist is set, then `CZ2` is interpreted as the co-moving distance, rather than *cz*.

  Must be of same precision type as RA1/DEC1/CZ1.

- **weights2** (*array-like, real (float/double), optional*) – Same as weights1, but for the second set of positions

- **is_comoving_dist** (*boolean (default false)*) – Boolean flag to indicate that `cz` values have already been converted into co-moving distances. This flag allows arbitrary cosmologies to be used in `Corrfunc`.

- **verbose** (*boolean (default false)*) – Boolean flag to control output of informational messages

- **output_savg** (*boolean (default false)*) – Boolean flag to output the average `s` for each bin. Code will run slower if you set this flag. Also, note, if you are calculating in single-precision, `savg` will suffer from numerical loss of precision and can not be trusted. If you need accurate `savg` values, then pass in double precision arrays for the particle positions.

- **fast_divide_and_NR_steps** (*integer (default 0)*) – Replaces the division in `AVX` implementation with an approximate reciprocal, followed by `fast_divide_and_NR_steps` of Newton-Raphson. Can improve runtime by ~15-20% on older computers. Value of 0 uses the standard division operation.

- **(xyz)bin_refine_factor** (*integer, default is (2,2,1); typically within [1-3]*) – Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

- **max_cells_per_dim** (*integer, default is 100, typical values in [50-300]*) – Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rpmax` is too small relative to the boxsize (and increasing helps the runtime).

- **c_api_timer** (*boolean (default false)*) – Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

- **isa** (string (default `fastest`)) – Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

  Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

  Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

- **weight_type** (*string, optional*) – The type of weighting to apply. One of ["pair_product", None]. Default: None.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [smin, smax, savg, mumax, npairs, weightavg] for each separation bin specified in the `binfile`. If `output_savg` is not set, then `savg` will be set to 0.0 for all bins; similarly for

`weightavg`. `npairs` contains the number of pairs in that bin and can be used to compute the actual $\xi(s, \mu)$ by combining with (DR, RR) counts.

- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Corrfunc.mocks.DDtheta_mocks module

Python wrapper around the C extension for the angular correlation function $\omega(\theta)$. Corresponding C routines are in `mocks/DDtheta_mocks/`, while the python interface is *Corrfunc.mocks.DDtheta_mocks*

Corrfunc.mocks.DDtheta_mocks.**DDtheta_mocks**(*autocorr*, *nthreads*, *binfile*, *RA1*, *DEC1*, *weights1=None*, *RA2=None*, *DEC2=None*, *weights2=None*, *link_in_dec=True*, *link_in_ra=True*, *verbose=False*, *output_thetaavg=False*, *fast_acos=False*, *ra_refine_factor=2*, *dec_refine_factor=2*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*, *weight_type=None*)

Function to compute the angular correlation function for points on the sky (i.e., mock catalogs or observed galaxies).

Returns a numpy structured array containing the pair counts for the specified angular bins.

If `weights` are provided, the resulting pair counts are weighted. The weighting scheme depends on `weight_type`.

Note: This module only returns pair counts and not the actual correlation function $\omega(heta)$. See *Corrfunc.utils.convert_3d_counts_to_cf* for computing $\omega(heta)$ from the pair counts returned.

Parameters

- **autocorr** (*boolean, required*) – Boolean flag for auto/cross-correlation. If autocorr is set to 1, then the second set of particle positions are not required.
- **nthreads** (*integer*) – Number of threads to use.
- **binfile** (*string or an list/array of floats. Units: degrees.*) – For string input: filename specifying the `theta` bins for `DDtheta_mocks`. The file should contain white-space separated values of (thetamin, thetamax) for each `theta` wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

  For array-like input: A sequence of `theta` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0 degrees. This array does not need to be sorted.

- **RA1** (*array-like, real (float/double)*) – The array of Right Ascensions for the first set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

  Calculations are done in the precision of the supplied arrays.

- **DEC1** (*array-like, real (float/double)*) – Array of Declinations for the first set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases

where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0]. Must be of same precision type as RA1.

- **weights1** (*array_like, real (float/double), optional*) – A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field. If only one of weights1 and weights2 is specified, the other will be set to uniform weights.
- **RA2** (*array-like, real (float/double)*) – The array of Right Ascensions for the second set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0]. Must be of same precision type as RA1/DEC1.
- **DEC2** (*array-like, real (float/double)*) – Array of Declinations for the second set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0]. Must be of same precision type as RA1/DEC1.
- **weights2** (*array-like, real (float/double), optional*) – Same as weights1, but for the second set of positions
- **link_in_dec** (*boolean (default True)*) – Boolean flag to create lattice in Declination. Code runs faster with this option. However, if the angular separations are too small, then linking in declination might produce incorrect results. When running for the first time, check your results by comparing with the output of the code for `link_in_dec=False` and `link_in_ra=False`.
- **link_in_ra** (*boolean (default True)*) – Boolean flag to create lattice in Right Ascension. Setting this option implies `link_in_dec=True`. Similar considerations as `link_in_dec` described above.

  If you disable both `link_in_dec` and `link_in_ra`, then the code reduces to a brute-force pair counter. No lattices are created at all. For very small angular separations, the brute-force method might be the most numerically stable method.
- **verbose** (*boolean (default false)*) – Boolean flag to control output of informational messages
- **output_thetaavg** (*boolean (default false)*) – Boolean flag to output the average '' heta'' for each bin. Code will run slower if you set this flag.

  If you are calculating in single-precision, `thetaavg` will suffer from numerical loss of precision and can not be trusted. If you need accurate `thetaavg` values, then pass in double precision arrays for `RA/DEC`.

  Code will run significantly slower if you enable this option. Use the keyword `fast_acos` if you can tolerate some loss of precision.
- **fast_acos** (*boolean (default false)*) – Flag to use numerical approximation for the `arccos` - gives better performance at the expense of some precision. Relevant only if `output_thetaavg==True`.

  Developers: Two versions already coded up in `utils/fast_acos.h`, so you can choose the version you want. There are also notes on how to implement faster (and less accurate) functions, particularly relevant if you know your `theta` range is limited. If you implement a new version, then you will have to reinstall the entire Corrfunc package.

  Note: Tests will fail if you run the tests with''fast_acos=True''.
- **(radec)_refine_factor** (*integer, default is (2,2); typically within [1-3]*) – Controls the refinement on the cell sizes. Can have up to a 20% impact

on runtime.

Only two refine factors are to be specified and these correspond to `ra` and `dec` (rather, than the usual three of `(xyz)bin_refine_factor` for all other correlation functions).

- **max_cells_per_dim** (*integer, default is 100, typical values in [50-300]*) – Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `thetamax` is too small relative to the boxsize (and increasing helps the runtime).
- **c_api_timer** (*boolean (default false)*) – Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.
- **isa** (string (default `fastest`)) – Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

  Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

  Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [thetamin, thetamax, thetaavg, npairs, weightavg] for each angular bin specified in the `binfile`. If `output_thetaavg` is not set then `thetavg` will be set to 0.0 for all bins; similarly for `weightavg`. `npairs` contains the number of pairs in that bin.
- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> import time
>>> from math import pi
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.mocks.DDtheta_mocks import DDtheta_mocks
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../mocks/tests/", "angular_bins")
>>> N = 100000
>>> nthreads = 4
>>> seed = 42
>>> np.random.seed(seed)
>>> RA = np.random.uniform(0.0, 2.0*pi, N)*180.0/pi
>>> cos_theta = np.random.uniform(-1.0, 1.0, N)
>>> DEC = 90.0 - np.arccos(cos_theta)*180.0/pi
>>> weights = np.ones_like(RA)
>>> autocorr = 1
>>> for isa in ['AVX', 'SSE42', 'FALLBACK']:
...     for link_in_dec in [False, True]:
...         for link_in_ra in [False, True]:
...             results = DDtheta_mocks(autocorr, nthreads, binfile,
```

(continues on next page)

(continued from previous page)

```
...                          RA, DEC, output_thetaavg=True,
...                          weights1=weights, weight_type='pair_product',
...                          link_in_dec=link_in_dec, link_in_ra=link_in_ra,
...                          isa=isa, verbose=True)
>>> for r in results: print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10d} {4:10.6f}".
...                          format(r['thetamin'], r['thetamax'],
...                          r['thetaavg'], r['npairs'], r['weightavg']))
...
  0.010000   0.014125   0.012272         62   1.000000
  0.014125   0.019953   0.016978        172   1.000000
  0.019953   0.028184   0.024380        298   1.000000
  0.028184   0.039811   0.034321        598   1.000000
  0.039811   0.056234   0.048535       1164   1.000000
  0.056234   0.079433   0.068385       2438   1.000000
  0.079433   0.112202   0.096631       4658   1.000000
  0.112202   0.158489   0.136834       9414   1.000000
  0.158489   0.223872   0.192967      19098   1.000000
  0.223872   0.316228   0.272673      37848   1.000000
  0.316228   0.446684   0.385344      75520   1.000000
  0.446684   0.630957   0.543973     150938   1.000000
  0.630957   0.891251   0.768406     301854   1.000000
  0.891251   1.258925   1.085273     599896   1.000000
  1.258925   1.778279   1.533461    1200238   1.000000
  1.778279   2.511886   2.166009    2396338   1.000000
  2.511886   3.548134   3.059159    4775162   1.000000
  3.548134   5.011872   4.321445    9532582   1.000000
  5.011872   7.079458   6.104214   19001930   1.000000
  7.079458  10.000000   8.622400   37842502   1.000000
```

Corrfunc.mocks.vpf_mocks module

Python wrapper around the C extension for the counts-in-cells for positions on the sky. Corresponding C codes are in `mocks/vpf_mocks/` while the python wrapper is in *Corrfunc.mocks.vpf_mocks*

`Corrfunc.mocks.vpf_mocks.`**`vpf_mocks`**(*rmax*, *nbins*, *nspheres*, *numpN*, *threshold_ngb*, *centers_file*, *cosmology*, *RA*, *DEC*, *CZ*, *RAND_RA*, *RAND_DEC*, *RAND_CZ*, *verbose=False*, *is_comoving_dist=False*, *xbin_refine_factor=1*, *ybin_refine_factor=1*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*)

Function to compute the counts-in-cells on points on the sky. Suitable for mock catalogs and observed galaxies.

Returns a numpy structured array containing the probability of a sphere of radius up to `rmax` containing `0--numpN-1` galaxies.

Parameters

- **`rmax`** (*`double`*) – Maximum radius of the sphere to place on the particles
- **`nbins`** (*`integer`*) – Number of bins in the counts-in-cells. Radius of first shell is rmax/nbins
- **`nspheres`** (*`integer (>= 0)`*) – Number of random spheres to place within the particle distribution. For a small number of spheres, the error is larger in the measured pN's.
- **`numpN`** (*`integer (>= 1)`*) – Governs how many unique pN's are to returned. If `numpN` is set to 1, then only the vpf (p0) is returned. For `numpN=2`, p0 and p1 are returned.

More explicitly, the columns in the results look like the following:

numpN	Columns in output
1	p0
2	p0 p1
3	p0 p1 p2
4	p0 p1 p2 p3

and so on...

Note: `p0` is the vpf

- **`threshold_ngb`** (*`integer`*) – Minimum number of random points needed in a `rmax` sphere such that it is considered to be entirely within the mock footprint. The command-line version, `mocks/vpf/vpf_mocks.c`, assumes that the minimum number of randoms can be at most a 1-sigma deviation from the expected random number density.

- **`centers_file`** (*`string, filename`*) – A file containing random sphere centers. If the file does not exist, then a list of random centers will be written out. In that case, the randoms arrays, `RAND_RA`, `RAND_DEC` and `RAND_CZ` are used to check that the sphere is entirely within the footprint. If the file does exist but either `rmax` is too small or there are not enough centers then the file will be overwritten.

  Note: If the centers file has to be written, the code will take significantly longer to finish. However, subsequent runs can re-use that centers file and will be faster.

- **`cosmology`** (*`integer, required`*) – Integer choice for setting cosmology. Valid values are 1->LasDamas cosmology and 2->Planck cosmology. If you need arbitrary cosmology, easiest way is to convert the `CZ` values into co-moving distance, based on your preferred cosmology. Set `is_comoving_dist=True`, to indicate that the co-moving distance conversion has already been done.

  Choices:

  1. LasDamas cosmology. $\Omega_m = 0.25$, $\Omega_\Lambda = 0.75$
  2. Planck cosmology. $\Omega_m = 0.302$, $\Omega_\Lambda = 0.698$

  To setup a new cosmology, add an entry to the function, `init_cosmology` in `ROOT/utils/cosmology_params.c` and re-install the entire package.

- **`RA`** (*`array-like, real (float/double)`*) – The array of Right Ascensions for the first set of points. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

  Calculations are done in the precision of the supplied arrays.

- **`DEC`** (*`array-like, real (float/double)`*) – Array of Declinations for the first set of points. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0].

  Must be of same precision type as RA.

- **`CZ`** (*`array-like, real (float/double)`*) – Array of (Speed Of Light * Redshift) values for the first set of points. Code will try to detect cases where `redshifts` have been passed and multiply the entire array with the `speed of light`.

  If `is_comoving_dist` is set, then `CZ` is interpreted as the co-moving distance, rather than (Speed Of Light * Redshift).

- **RAND_RA** (*array-like, real (float/double)*) – The array of Right Ascensions for the randoms. RA's are expected to be in [0.0, 360.0], but the code will try to fix cases where the RA's are in [-180, 180.0]. For peace of mind, always supply RA's in [0.0, 360.0].

  Must be of same precision type as RA/DEC/CZ.

- **RAND_DEC** (*array-like, real (float/double)*) – Array of Declinations for the randoms. DEC's are expected to be in the [-90.0, 90.0], but the code will try to fix cases where the DEC's are in [0.0, 180.0]. Again, for peace of mind, always supply DEC's in [-90.0, 90.0].

  Must be of same precision type as RA/DEC/CZ.

- **RAND_CZ** (*array-like, real (float/double)*) – Array of (Speed Of Light * Redshift) values for the randoms. Code will try to detect cases where `redshifts` have been passed and multiply the entire array with the `speed of light`.

  If `is_comoving_dist` is set, then `CZ2` is interpreted as the co-moving distance, rather than `(Speed Of Light * Redshift)`.

  Note: RAND_RA, RAND_DEC and RAND_CZ are only used when the `centers_file` needs to be written out. In that case, the RAND_RA, RAND_DEC, and RAND_CZ are used as random centers.

- **verbose** (*boolean (default false)*) – Boolean flag to control output of informational messages

- **is_comoving_dist** (*boolean (default false)*) – Boolean flag to indicate that `cz` values have already been converted into co-moving distances. This flag allows arbitrary cosmologies to be used in `Corrfunc`.

- **(xyz)bin_refine_factor** (*integer, default is (1,1,1); typically within [1-3]*) – Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

  Note: Since the counts in spheres calculation is symmetric in all 3 dimensions, the defaults are different from the clustering routines.

- **max_cells_per_dim** (*integer, default is 100, typical values in [50-300]*) – Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rmax` is too small relative to the boxsize (and increasing helps the runtime).

- **c_api_timer** (*boolean (default false)*) – Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

- **isa** (string (default `fastest`)) – Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

  Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

  Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [rmax, pN[numpN]] with `nbins` elements. Each row contains the maximum radius of the sphere

and the `numpN` elements in the `pN` array. Each element of this array contains the probability that a sphere of radius `rmax` contains *exactly* `N` galaxies. For example, pN[0] (p0, the void probibility function) is the probability that a sphere of radius `rmax` contains 0 galaxies.

- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Example

```
>>> from __future__ import print_function
>>> import math
>>> from os.path import dirname, abspath, join as pjoin
>>> import numpy as np
>>> import Corrfunc
>>> from Corrfunc.mocks.vpf_mocks import vpf_mocks
>>> rmax = 10.0
>>> nbins = 10
>>> numbins_to_print = nbins
>>> nspheres = 10000
>>> numpN = 6
>>> threshold_ngb = 1  # does not matter since we have the centers
>>> cosmology = 1  # LasDamas cosmology
>>> centers_file = pjoin(dirname(abspath(Corrfunc.__file__)),
...                      "../mocks/tests/data/",
...                      "Mr19_centers_xyz_forVPF_rmax_10Mpc.txt")
>>> N = 1000000
>>> boxsize = 420.0
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(-0.5*boxsize, 0.5*boxsize, N)
>>> Y = np.random.uniform(-0.5*boxsize, 0.5*boxsize, N)
>>> Z = np.random.uniform(-0.5*boxsize, 0.5*boxsize, N)
>>> CZ = np.sqrt(X*X + Y*Y + Z*Z)
>>> inv_cz = 1.0/CZ
>>> X *= inv_cz
>>> Y *= inv_cz
>>> Z *= inv_cz
>>> DEC = 90.0 - np.arccos(Z)*180.0/math.pi
>>> RA = (np.arctan2(Y, X)*180.0/math.pi) + 180.0
>>> results = vpf_mocks(rmax, nbins, nspheres, numpN, threshold_ngb,
...                     centers_file, cosmology,
...                     RA, DEC, CZ,
...                     RA, DEC, CZ,
...                     is_comoving_dist=True)
>>> for r in results:
...     print("{0:10.1f} ".format(r[0]), end="")
...
...     for pn in r[1]:
...         print("{0:10.3f} ".format(pn), end="")
...
...     print("")
   1.0      0.999      0.001      0.000      0.000      0.000      0.000
   2.0      0.992      0.007      0.001      0.000      0.000      0.000
   3.0      0.982      0.009      0.005      0.002      0.001      0.000
   4.0      0.975      0.006      0.006      0.005      0.003      0.003
   5.0      0.971      0.004      0.003      0.003      0.004      0.003
```

(continues on next page)

(continued from previous page)

6.0	0.967	0.003	0.003	0.001	0.003	0.002
7.0	0.962	0.004	0.002	0.003	0.002	0.001
8.0	0.958	0.004	0.002	0.003	0.001	0.002
9.0	0.953	0.003	0.003	0.002	0.003	0.001
10.0	0.950	0.003	0.002	0.002	0.001	0.002

Corrfunc.theory package

Wrapper for all clustering statistic calculations on galaxies in a simulation volume.

Corrfunc.theory.**DD**(*autocorr*, *nthreads*, *binfile*, *X1*, *Y1*, *Z1*, *weights1=None*, *periodic=True*, *X2=None*, *Y2=None*, *Z2=None*, *weights2=None*, *verbose=False*, *boxsize=0.0*, *output_ravg=False*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*, *weight_type=None*)

Calculate the 3-D pair-counts corresponding to the real-space correlation function, $\xi(r)$.

If `weights` are provided, the resulting pair counts are weighted. The weighting scheme depends on `weight_type`.

Note: This module only returns pair counts and not the actual correlation function $\xi(r)$. See *Corrfunc.utils.convert_3d_counts_to_cf* for computing for computing $\xi(r)$ from the pair counts returned.

Parameters

- **autocorr** (*boolean, required*) – Boolean flag for auto/cross-correlation. If autocorr is set to 1, then the second set of particle positions are not required.
- **nthreads** (*integer*) – The number of OpenMP threads to use. Has no effect if OpenMP was not enabled during library compilation.
- **binfile** (*string or an list/array of floats*) – For string input: filename specifying the `r` bins for `DD`. The file should contain white-space separated values of (rmin, rmax) for each `r` wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

  For array-like input: A sequence of `r` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.
- **X1/Y1/Z1** (*array_like, real (float/double)*) – The array of X/Y/Z positions for the first set of points. Calculations are done in the precision of the supplied arrays.
- **weights1** (*array_like, real (float/double), optional*) – A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field. If only one of weights1 and weights2 is specified, the other will be set to uniform weights.
- **periodic** (*boolean*) – Boolean flag to indicate periodic boundary conditions.
- **X2/Y2/Z2** (*array-like, real (float/double)*) – Array of XYZ positions for the second set of points. *Must* be the same precision as the X1/Y1/Z1 arrays. Only required when `autocorr==0`.
- **weights2** (*array-like, real (float/double), optional*) – Same as weights1, but for the second set of positions

- **verbose** (*boolean (default false)*) – Boolean flag to control output of informational messages
- **boxsize** (*double*) – The side-length of the cube in the cosmological simulation. Present to facilitate exact calculations for periodic wrapping. If boxsize is not supplied, then the wrapping is done based on the maximum difference within each dimension of the X/Y/Z arrays.
- **output_ravg** (*boolean (default false)*) – Boolean flag to output the average `r` for each bin. Code will run slower if you set this flag.

  Note: If you are calculating in single-precision, `ravg` will suffer from numerical loss of precision and can not be trusted. If you need accurate `ravg` values, then pass in double precision arrays for the particle positions.

(xyz)bin_refine_factor: integer, default is (2,2,1); typically within [1-3] Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

max_cells_per_dim: integer, default is 100, typical values in [50-300] Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rmax` is too small relative to the boxsize (and increasing helps the runtime).

c_api_timer: boolean (default false) Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

isa: string (default `fastest`) Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

weight_type: string, optional The type of weighting to apply. One of ["pair_product", None]. Default: None.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [rmin, rmax, ravg, npairs, weightavg] for each radial bin specified in the `binfile`. If `output_ravg` is not set, then `ravg` will be set to 0.0 for all bins; similarly for `weightavg`. `npairs` contains the number of pairs in that bin and can be used to compute the actual $\xi(r)$ by combining with (DR, RR) counts.
- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.theory.DD import DD
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../theory/tests/", "bins")
```

(continues on next page)

(continued from previous page)

```
>>> N = 10000
>>> boxsize = 420.0
>>> nthreads = 4
>>> autocorr = 1
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(0, boxsize, N)
>>> Y = np.random.uniform(0, boxsize, N)
>>> Z = np.random.uniform(0, boxsize, N)
>>> weights = np.ones_like(X)
>>> results = DD(autocorr, nthreads, binfile, X, Y, Z, weights1=weights, weight_
↪type='pair_product', output_ravg=True)
>>> for r in results: print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10d} {4:10.6f}".
...                         format(r['rmin'], r['rmax'], r['ravg'],
...                         r['npairs'], r['weightavg']))
  0.167536   0.238755   0.000000          0   0.000000
  0.238755   0.340251   0.000000          0   0.000000
  0.340251   0.484892   0.000000          0   0.000000
  0.484892   0.691021   0.000000          0   0.000000
  0.691021   0.984777   0.945372          2   1.000000
  0.984777   1.403410   1.340525         10   1.000000
  1.403410   2.000000   1.732968         36   1.000000
  2.000000   2.850200   2.558878         54   1.000000
  2.850200   4.061840   3.564959        208   1.000000
  4.061840   5.788530   4.999278        674   1.000000
  5.788530   8.249250   7.126673       2154   1.000000
  8.249250  11.756000  10.201834       5996   1.000000
 11.756000  16.753600  14.517830      17746   1.000000
 16.753600  23.875500  20.716017      50252   1.000000
```

Corrfunc.theory.**DDrppi**(*autocorr*, *nthreads*, *pimax*, *binfile*, *X1*, *Y1*, *Z1*, *weights1=None*, *periodic=True*, *X2=None*, *Y2=None*, *Z2=None*, *weights2=None*, *verbose=False*, *boxsize=0.0*, *output_rpavg=False*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*, *weight_type=None*)

Calculate the 3-D pair-counts corresponding to the real-space correlation function, $\xi(r_p, \pi)$ or $\wp(r_p)$. Pairs which are separated by less than the `rp` bins (specified in `binfile`) in the X-Y plane, and less than `pimax` in the Z-dimension are counted.

If `weights` are provided, the resulting pair counts are weighted. The weighting scheme depends on `weight_type`.

Note: that this module only returns pair counts and not the actual correlation function $\xi(r_p, \pi)$ or $wp(r_p)$. See the utilities *Corrfunc.utils.convert_3d_counts_to_cf* and *Corrfunc.utils.convert_rp_pi_counts_to_wp* for computing $\xi(r_p, \pi)$ and $wp(r_p)$ respectively from the pair counts.

Parameters

- **autocorr** (*boolean, required*) – Boolean flag for auto/cross-correlation. If autocorr is set to 1, then the second set of particle positions are not required.
- **nthreads** (*integer*) – The number of OpenMP threads to use. Has no effect if OpenMP was not enabled during library compilation.
- **pimax** (*double*) – A double-precision value for the maximum separation along the Z-dimension.

Distances along the :math:\pi direction are binned with unit depth. For instance, if pimax=40, then 40 bins will be created along the pi direction.

Note: Only pairs with 0 <= dz < pimax are counted (no equality).

binfile: string or an list/array of floats For string input: filename specifying the rp bins for DDrppi. The file should contain white-space separated values of (rpmin, rpmax) for each rp wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

For array-like input: A sequence of rp values that provides the bin-edges. For example, np.logspace(np.log10(0.1), np.log10(10.0), 15) is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.

X1/Y1/Z1: array-like, real (float/double) The array of X/Y/Z positions for the first set of points. Calculations are done in the precision of the supplied arrays.

weights1: array_like, real (float/double), optional A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field. If only one of weights1 and weights2 is specified, the other will be set to uniform weights.

X2/Y2/Z2: array-like, real (float/double) Array of XYZ positions for the second set of points. *Must* be the same precision as the X1/Y1/Z1 arrays. Only required when autocorr==0.

weights2: array-like, real (float/double), optional Same as weights1, but for the second set of positions

periodic: boolean Boolean flag to indicate periodic boundary conditions.

verbose: boolean (default false) Boolean flag to control output of informational messages

boxsize: double The side-length of the cube in the cosmological simulation. Present to facilitate exact calculations for periodic wrapping. If boxsize is not supplied, then the wrapping is done based on the maximum difference within each dimension of the X/Y/Z arrays.

output_rpavg: boolean (default false) Boolean flag to output the average rp for each bin. Code will run slower if you set this flag.

Note: If you are calculating in single-precision, rpavg will suffer from numerical loss of precision and can not be trusted. If you need accurate rpavg values, then pass in double precision arrays for the particle positions.

(xyz)bin_refine_factor: integer, default is (2,2,1); typically within [1-3] Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

max_cells_per_dim: integer, default is 100, typical values in [50-300] Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if rpmax is too small relative to the boxsize (and increasing helps the runtime).

c_api_timer: boolean (default false) Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

isa: string (default `fastest`) Controls the runtime dispatch for the instruction set to use. Possible options are: [fastest, avx, sse42, fallback]

Setting isa to fastest will pick the fastest available instruction set on the current computer. However, if you set isa to, say, avx and avx is not available on the computer, then the code will revert to using fallback (even though sse42 might be available).

Unless you are benchmarking the different instruction sets, you should always leave isa to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an enum for the instruction set defined in utils/defs.h.

weight_type: string, optional The type of weighting to apply. One of ["pair_product", None]. Default: None.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [rpmin, rpmax, rpavg, pimax, npairs, weightavg] for each radial bin specified in the `binfile`. If `output_rpavg` is not set, then `rpavg` will be set to 0.0 for all bins; similarly for `weightavg`. `npairs` contains the number of pairs in that bin and can be used to compute $\xi(r_p, \pi)$ by combining with (DR, RR) counts.
- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.theory.DDrppi import DDrppi
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../theory/tests/", "bins")
>>> N = 10000
>>> boxsize = 420.0
>>> nthreads = 4
>>> autocorr = 1
>>> pimax = 40.0
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(0, boxsize, N)
>>> Y = np.random.uniform(0, boxsize, N)
>>> Z = np.random.uniform(0, boxsize, N)
>>> weights = np.ones_like(X)
>>> results = DDrppi(autocorr, nthreads, pimax, binfile,
...                  X, Y, Z, weights1=weights, weight_type='pair_product',␣
↪output_rpavg=True)
>>> for r in results[519:]: print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10.1f}"
...                               " {4:10d} {5:10.6f}".format(r['rmin'], r['rmax
↪'],
...                               r['rpavg'], r['pimax'], r['npairs'], r[
↪'weightavg']))
...
 11.756000  16.753600  14.379250       40.0       1150   1.000000
 16.753600  23.875500  20.449131        1.0       2604   1.000000
 16.753600  23.875500  20.604834        2.0       2370   1.000000
 16.753600  23.875500  20.523989        3.0       2428   1.000000
 16.753600  23.875500  20.475181        4.0       2462   1.000000
 16.753600  23.875500  20.458005        5.0       2532   1.000000
 16.753600  23.875500  20.537162        6.0       2522   1.000000
 16.753600  23.875500  20.443087        7.0       2422   1.000000
 16.753600  23.875500  20.474580        8.0       2360   1.000000
 16.753600  23.875500  20.420360        9.0       2512   1.000000
 16.753600  23.875500  20.478355       10.0       2472   1.000000
 16.753600  23.875500  20.485268       11.0       2406   1.000000
 16.753600  23.875500  20.372985       12.0       2420   1.000000
 16.753600  23.875500  20.647998       13.0       2378   1.000000
 16.753600  23.875500  20.556208       14.0       2420   1.000000
 16.753600  23.875500  20.527992       15.0       2462   1.000000
 16.753600  23.875500  20.581017       16.0       2380   1.000000
```

(continues on next page)

(continued from previous page)

```
 16.753600  23.875500  20.491819       17.0       2346   1.000000
 16.753600  23.875500  20.534440       18.0       2496   1.000000
 16.753600  23.875500  20.529129       19.0       2512   1.000000
 16.753600  23.875500  20.501946       20.0       2500   1.000000
 16.753600  23.875500  20.513349       21.0       2544   1.000000
 16.753600  23.875500  20.471915       22.0       2430   1.000000
 16.753600  23.875500  20.450651       23.0       2354   1.000000
 16.753600  23.875500  20.550753       24.0       2460   1.000000
 16.753600  23.875500  20.540262       25.0       2490   1.000000
 16.753600  23.875500  20.559572       26.0       2350   1.000000
 16.753600  23.875500  20.534245       27.0       2382   1.000000
 16.753600  23.875500  20.511302       28.0       2508   1.000000
 16.753600  23.875500  20.491632       29.0       2456   1.000000
 16.753600  23.875500  20.592493       30.0       2386   1.000000
 16.753600  23.875500  20.506234       31.0       2484   1.000000
 16.753600  23.875500  20.482109       32.0       2538   1.000000
 16.753600  23.875500  20.518463       33.0       2544   1.000000
 16.753600  23.875500  20.482515       34.0       2534   1.000000
 16.753600  23.875500  20.503124       35.0       2382   1.000000
 16.753600  23.875500  20.471307       36.0       2356   1.000000
 16.753600  23.875500  20.384231       37.0       2554   1.000000
 16.753600  23.875500  20.454012       38.0       2458   1.000000
 16.753600  23.875500  20.585543       39.0       2394   1.000000
 16.753600  23.875500  20.504965       40.0       2500   1.000000
```

Corrfunc.theory.**wp**(*boxsize*, *pimax*, *nthreads*, *binfile*, *X*, *Y*, *Z*, *weights=None*, *weight_type=None*, *verbose=False*, *output_rpavg=False*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *c_cell_timer=False*, *isa=u'fastest'*)

Function to compute the projected correlation function in a periodic cosmological box. Pairs which are separated by less than the `rp` bins (specified in `binfile`) in the X-Y plane, and less than `pimax` in the Z-dimension are counted.

If `weights` are provided, the resulting correlation function is weighted. The weighting scheme depends on `weight_type`.

Note: Pairs are double-counted. And if `rpmin` is set to 0.0, then all the self-pairs (i'th particle with itself) are added to the first bin => minimum number of pairs in the first bin is the total number of particles.

Parameters

- **boxsize** (*double*) – A double-precision value for the boxsize of the simulation in same units as the particle positions and the `rp` bins.
- **pimax** (*double*) – A double-precision value for the maximum separation along the Z-dimension.

  Note: Only pairs with `0 <= dz < pimax` are counted (no equality).

nthreads: integer Number of threads to use.

binfile: string or an list/array of floats For string input: filename specifying the `rp` bins for `wp`. The file should contain white-space separated values of (rpmin, rpmax) for each `rp` wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

For array-like input: A sequence of `rp` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.

X/Y/Z: arraytype, real (float/double) Particle positions in the 3 axes. Must be within [0, boxsize] and specified in the same units as `rp_bins` and boxsize. All 3 arrays must be of the same floating-point type.

Calculations will be done in the same precision as these arrays, i.e., calculations will be in floating point if XYZ are single precision arrays (C float type); or in double-precision if XYZ are double precision arrays (C double type).

weights: array_like, real (float/double), optional A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field.

verbose: boolean (default false) Boolean flag to control output of informational messages

output_rpavg: boolean (default false) Boolean flag to output the average `rp` for each bin. Code will run slower if you set this flag.

Note: If you are calculating in single-precision, `rpavg` will suffer from numerical loss of precision and can not be trusted. If you need accurate `rpavg` values, then pass in double precision arrays for the particle positions.

(xyz)bin_refine_factor: integer, default is (2,2,1); typically within [1-3] Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

max_cells_per_dim: integer, default is 100, typical values in [50-300] Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rpmax` is too small relative to the boxsize (and increasing helps the runtime).

c_api_timer: boolean (default false) Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

c_cell_timer [boolean (default false)] Boolean flag to measure actual time spent **per cell-pair** within the C libraries. A very detailed timer that stores information about the number of particles in each cell, the thread id that processed that cell-pair and the amount of time in nano-seconds taken to process that cell pair. This timer can be used to study the instruction set efficiency, and load-balancing of the code.

isa: string (default `fastest`) Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

weight_type: string, optional The type of weighting to apply. One of ["pair_product", None]. Default: None.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [rpmin, rpmax, rpavg, wp, npairs, weightavg] for each radial specified in the `binfile`. If `output_rpavg` is not set then `rpavg` will be set to 0.0 for all bins; similarly for `weightavg`. `wp` contains the projected correlation function while `npairs` contains the number of unique pairs in that bin. If using weights, `wp` will be weighted while `npairs` will not be.

- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.
- **cell_time** (*list, optional*) – Only returned if `c_cell_timer` is set. Contains detailed stats about each cell-pair visited during pair-counting, viz., number of particles in each of the cells in the pair, 1-D cell-indices for each cell in the pair, time (in nano-seconds) to process the pair and the thread-id for the thread that processed that cell-pair.

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.theory.wp import wp
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../theory/tests/", "bins")
>>> N = 10000
>>> boxsize = 420.0
>>> pimax = 40.0
>>> nthreads = 4
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(0, boxsize, N)
>>> Y = np.random.uniform(0, boxsize, N)
>>> Z = np.random.uniform(0, boxsize, N)
>>> results = wp(boxsize, pimax, nthreads, binfile, X, Y, Z, weights=np.ones_
↪like(X), weight_type='pair_product')
>>> for r in results:
...     print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10.6f} {4:10d} {5:10.6f}".
...           format(r['rmin'], r['rmax'],
...           r['rpavg'], r['wp'], r['npairs'], r['weightavg']))
...
  0.167536   0.238755   0.000000  66.717143         18   1.000000
  0.238755   0.340251   0.000000 -15.786045         16   1.000000
  0.340251   0.484892   0.000000   2.998470         42   1.000000
  0.484892   0.691021   0.000000 -15.779885         66   1.000000
  0.691021   0.984777   0.000000 -11.966728        142   1.000000
  0.984777   1.403410   0.000000  -9.699906        298   1.000000
  1.403410   2.000000   0.000000 -11.698771        588   1.000000
  2.000000   2.850200   0.000000   3.848375       1466   1.000000
  2.850200   4.061840   0.000000  -0.921452       2808   1.000000
  4.061840   5.788530   0.000000   0.454851       5802   1.000000
  5.788530   8.249250   0.000000   1.428344      11926   1.000000
  8.249250  11.756000   0.000000  -1.067885      23478   1.000000
 11.756000  16.753600   0.000000  -0.553319      47994   1.000000
 16.753600  23.875500   0.000000  -0.086433      98042   1.000000
```

`Corrfunc.theory.`**`xi`**(*boxsize*, *nthreads*, *binfile*, *X*, *Y*, *Z*, *weights=None*, *weight_type=None*, *verbose=False*, *output_ravg=False*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*)

Function to compute the projected correlation function in a periodic cosmological box. Pairs which are separated by less than the `r` bins (specified in `binfile`) in 3-D real space.

If `weights` are provided, the resulting correlation function is weighted. The weighting scheme depends on `weight_type`.

Note: Pairs are double-counted. And if `rmin` is set to 0.0, then all the self-pairs (i'th particle with itself) are added to the first bin => minimum number of pairs in the first bin is the total number of particles.

Parameters

- **`boxsize`** (*`double`*) – A double-precision value for the boxsize of the simulation in same units as the particle positions and the `r` bins.
- **`nthreads`** (*`integer`*) – Number of threads to use.
- **`binfile`** (*`string or an list/array of floats`*) – For string input: filename specifying the `r` bins for `xi`. The file should contain white-space separated values of (rmin, rmax) for each `r` wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

  For array-like input: A sequence of `r` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.
- **`X/Y/Z`** (*`arraytype, real (float/double)`*) – Particle positions in the 3 axes. Must be within [0, boxsize] and specified in the same units as `rp_bins` and boxsize. All 3 arrays must be of the same floating-point type.

  Calculations will be done in the same precision as these arrays, i.e., calculations will be in floating point if XYZ are single precision arrays (C float type); or in double-precision if XYZ are double precision arrays (C double type).
- **`weights`** (*`array_like, real (float/double), optional`*) – A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field.
- **`verbose`** (*`boolean (default false)`*) – Boolean flag to control output of informational messages
- **`output_ravg`** (*`boolean (default false)`*) – Boolean flag to output the average `r` for each bin. Code will run slower if you set this flag.

  Note: If you are calculating in single-precision, `rpavg` will suffer from numerical loss of precision and can not be trusted. If you need accurate `rpavg` values, then pass in double precision arrays for the particle positions.

(xyz)bin_refine_factor: integer, default is (2,2,1); typically within [1-3] Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

max_cells_per_dim: integer, default is 100, typical values in [50-300] Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rmax` is too small relative to the boxsize (and increasing helps the runtime).

c_api_timer: boolean (default false) Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

isa: string (default `fastest`) Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

weight_type: string, optional, Default: None. The type of weighting to apply. One of ["pair_product", None].

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [rmin, rmax, ravg, xi, npairs, weightavg] for each radial specified in the `binfile`. If `output_ravg` is not set then `ravg` will be set to 0.0 for all bins; similarly for `weightavg`. `xi` contains the correlation function while `npairs` contains the number of pairs in that bin. If using weights, `xi` will be weighted while `npairs` will not be.
- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.theory.xi import xi
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../theory/tests/", "bins")
>>> N = 100000
>>> boxsize = 420.0
>>> nthreads = 4
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(0, boxsize, N)
>>> Y = np.random.uniform(0, boxsize, N)
>>> Z = np.random.uniform(0, boxsize, N)
>>> weights = np.ones_like(X)
>>> results = xi(boxsize, nthreads, binfile, X, Y, Z, weights=weights, weight_
↪type='pair_product', output_ravg=True)
>>> for r in results: print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10.6f} {4:10d}
↪{5:10.6f}"
...                      .format(r['rmin'], r['rmax'],
...                      r['ravg'], r['xi'], r['npairs'], r['weightavg']))
...
  0.167536   0.238755   0.226592  -0.205733         4   1.000000
  0.238755   0.340251   0.289277  -0.176729        12   1.000000
  0.340251   0.484892   0.426819  -0.051829        40   1.000000
  0.484892   0.691021   0.596187  -0.131853       106   1.000000
  0.691021   0.984777   0.850100  -0.049207       336   1.000000
  0.984777   1.403410   1.225112   0.028543      1052   1.000000
  1.403410   2.000000   1.737153   0.011403      2994   1.000000
  2.000000   2.850200   2.474588   0.005405      8614   1.000000
  2.850200   4.061840   3.532018  -0.014098     24448   1.000000
  4.061840   5.788530   5.022241  -0.010784     70996   1.000000
  5.788530   8.249250   7.160648  -0.001588    207392   1.000000
  8.249250  11.756000  10.207213  -0.000323    601002   1.000000
 11.756000  16.753600  14.541171   0.000007   1740084   1.000000
 16.753600  23.875500  20.728773  -0.001595   5028058   1.000000
```

Corrfunc.theory.**vpf**(*rmax*, *nbins*, *nspheres*, *numpN*, *seed*, *X*, *Y*, *Z*, *verbose=False*, *periodic=True*, *boxsize=0.0*, *xbin_refine_factor=1*, *ybin_refine_factor=1*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*)

Function to compute the counts-in-cells on 3-D real-space points.

Returns a numpy structured array containing the probability of a sphere of radius up to `rmax` containing [0, numpN-1] galaxies.

Parameters

- **rmax** (*double*) – Maximum radius of the sphere to place on the particles
- **nbins** (*integer*) – Number of bins in the counts-in-cells. Radius of first shell is rmax/nbins
- **nspheres** (*integer (>= 0)*) – Number of random spheres to place within the particle distribution. For a small number of spheres, the error is larger in the measured pN's.
- **numpN** (*integer (>= 1)*) – Governs how many unique pN's are to returned. If `numpN` is set to 1, then only the vpf (p0) is returned. For `numpN=2`, p0 and p1 are returned.

  More explicitly, the columns in the results look like the following:

numpN	Columns in output
1	p0
2	p0 p1
3	p0 p1 p2
4	p0 p1 p2 p3

  and so on...

  Note: `p0` is the vpf

seed: unsigned integer Random number seed for the underlying GSL random number generator. Used to draw centers of the spheres.

X/Y/Z: arraytype, real (float/double) Particle positions in the 3 axes. Must be within [0, boxsize] and specified in the same units as `rp_bins` and boxsize. All 3 arrays must be of the same floating-point type.

Calculations will be done in the same precision as these arrays, i.e., calculations will be in floating point if XYZ are single precision arrays (C float type); or in double-precision if XYZ are double precision arrays (C double type).

verbose: boolean (default false) Boolean flag to control output of informational messages

periodic: boolean Boolean flag to indicate periodic boundary conditions.

boxsize: double The side-length of the cube in the cosmological simulation. Present to facilitate exact calculations for periodic wrapping. If boxsize is not supplied, then the wrapping is done based on the maximum difference within each dimension of the X/Y/Z arrays.

(xyz)bin_refine_factor: integer, default is (1,1,1); typically within [1-3] Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

Note: Since the counts in spheres calculation is symmetric in all 3 dimensions, the defaults are different from the clustering routines.

max_cells_per_dim: integer, default is 100, typical values in [50-300] Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rmax` is too small relative to the boxsize (and increasing helps the runtime).

c_api_timer: boolean (default false) Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

isa: string (default `fastest`) Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

Returns

results – A numpy structured array containing [rmax, pN[numpN]] with `nbins` elements. Each row contains the maximum radius of the sphere and the `numpN` elements in the `pN` array. Each element of this array contains the probability that a sphere of radius `rmax` contains *exactly* `N` galaxies. For example, pN[0] (p0, the void probibility function) is the probability that a sphere of radius `rmax` contains 0 galaxies.

if `c_api_timer` is set, then the return value is a tuple containing (results, api_time). `api_time` measures only the time spent within the C library and ignores all python overhead.

Return type Numpy structured array

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from Corrfunc.theory.vpf import vpf
>>> rmax = 10.0
>>> nbins = 10
>>> nspheres = 10000
>>> numpN = 5
>>> seed = -1
>>> N = 100000
>>> boxsize = 420.0
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(0, boxsize, N)
>>> Y = np.random.uniform(0, boxsize, N)
>>> Z = np.random.uniform(0, boxsize, N)
>>> results = vpf(rmax, nbins, nspheres, numpN, seed, X, Y, Z)
>>> for r in results:
...     print("{0:10.1f} ".format(r[0]), end="")
...
...     for pn in r[1]:
...         print("{0:10.3f} ".format(pn), end="")
...
...     print("")
1.0       0.995      0.005      0.000      0.000      0.000
2.0       0.956      0.044      0.001      0.000      0.000
3.0       0.858      0.130      0.012      0.001      0.000
4.0       0.695      0.252      0.047      0.005      0.001
5.0       0.493      0.347      0.127      0.028      0.005
```

(continues on next page)

(continued from previous page)

```
6.0      0.295      0.362      0.219      0.091      0.026
7.0      0.141      0.285      0.265      0.179      0.085
8.0      0.056      0.159      0.228      0.229      0.161
9.0      0.019      0.066      0.135      0.192      0.192
10.0      0.003      0.019      0.054      0.106      0.150
```

Corrfunc.theory.**DDsmu**(*autocorr*, *nthreads*, *binfile*, *mu_max*, *nmu_bins*, *X1*, *Y1*, *Z1*, *weights1=None*, *periodic=True*, *X2=None*, *Y2=None*, *Z2=None*, *weights2=None*, *verbose=False*, *boxsize=0.0*, *output_savg=False*, *fast_divide_and_NR_steps=0*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*, *weight_type=None*)

Calculate the 2-D pair-counts corresponding to the redshift-space correlation function, $\xi(s, \mu)$ Pairs which are separated by less than the `s` bins (specified in `binfile`) in 3-D, and less than `s*mu_max` in the Z-dimension are counted.

If `weights` are provided, the resulting pair counts are weighted. The weighting scheme depends on `weight_type`.

Note: This module only returns pair counts and not the actual correlation function $\xi(s, \mu)$. See the utilities *Corrfunc.utils.convert_3d_counts_to_cf* for computing $\xi(s, \mu)$ from the pair counts.

New in version 2.1.0.

Parameters

- **autocorr** (*boolean, required*) – Boolean flag for auto/cross-correlation. If autocorr is set to 1, then the second set of particle positions are not required.
- **nthreads** (*integer*) – The number of OpenMP threads to use. Has no effect if OpenMP was not enabled during library compilation.
- **binfile** (*string or an list/array of floats*) – For string input: filename specifying the `s` bins for `DDsmu_mocks`. The file should contain white-space separated values of (smin, smax) specifying each `s` bin wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

  For array-like input: A sequence of `s` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.
- **mu_max** (*double. Must be in range (0.0, 1.0]*) – A double-precision value for the maximum cosine of the angular separation from the line of sight (LOS). Here, LOS is taken to be along the Z direction.

  Note: Only pairs with $0 <= \cos(\theta_{LOS}) < \mu_{max}$ are counted (no equality).
- **nmu_bins** (*int*) – The number of linear `mu` bins, with the bins ranging from from (0, μ_{max})
- **X1/Y1/Z1** (*array-like, real (float/double)*) – The array of X/Y/Z positions for the first set of points. Calculations are done in the precision of the supplied arrays.
- **weights1** (*array-like, real (float/double), shape (n_particles,) or (n_weights_per_particle,n_particles), optional*) – Weights for computing a weighted pair count.

- **weight_type** (*str, optional*) – The type of pair weighting to apply. Options: "pair_product", None; Default: None.
- **periodic** (*boolean*) – Boolean flag to indicate periodic boundary conditions.
- **X2/Y2/Z2** (*array-like, real (float/double)*) – Array of XYZ positions for the second set of points. *Must* be the same precision as the X1/Y1/Z1 arrays. Only required when `autocorr==0`.
- **weights2** (*array-like, real (float/double), shape (n_particles,) or (n_weights_per_particle,n_particles), optional*) – Weights for computing a weighted pair count.
- **verbose** (*boolean (default false)*) – Boolean flag to control output of informational messages
- **boxsize** (*double*) – The side-length of the cube in the cosmological simulation. Present to facilitate exact calculations for periodic wrapping. If boxsize is not supplied, then the wrapping is done based on the maximum difference within each dimension of the X/Y/Z arrays.
- **output_savg** (*boolean (default false)*) – Boolean flag to output the average `s` for each bin. Code will run slower if you set this flag. Also, note, if you are calculating in single-precision, `s` will suffer from numerical loss of precision and can not be trusted. If you need accurate `s` values, then pass in double precision arrays for the particle positions.
- **fast_divide_and_NR_steps** (*integer (default 0)*) – Replaces the division in `AVX` implementation with an approximate reciprocal, followed by `fast_divide_and_NR_steps` of Newton-Raphson. Can improve runtime by ~15-20% on older computers. Value of 0 uses the standard division operation.
- **(xyz)bin_refine_factor** (*integer (default (2,2,1) typical values in [1-3])*) – Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.
- **max_cells_per_dim** (*integer (default 100, typical values in [50-300])*) – Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rmax` is too small relative to the boxsize (and increasing helps the runtime).
- **c_api_timer** (*boolean (default false)*) – Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.
- **isa** (*integer (default -1)*) – Controls the runtime dispatch for the instruction set to use. Possible options are: [-1, AVX, SSE42, FALLBACK]

  Setting isa to -1 will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `AVX` and `AVX` is not available on the computer, then the code will revert to using `FALLBACK` (even though `SSE42` might be available).

  Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the integer values correspond to the `enum` for the instruction set defined in `utils/defs.h`.

Returns

- **results** (*A python list*) – A python list containing `nmu_bins` of [smin, smax, savg, mu_max, npairs, weightavg] for each spatial bin specified in the `binfile`. There will be a total of `nmu_bins` ranging from [0, `mu_max`) *per* spatial bin. If `output_savg` is not set, then `savg` will be set to 0.0 for all bins; similarly for `weight_avg`. `npairs` contains the number of pairs in that bin.

- **time** (if `c_api_timer` is set, then the return value contains the time spent) – in the API; otherwise time is set to 0.0

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.theory.DDsmu import DDsmu
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../theory/tests/", "bins")
>>> N = 10000
>>> boxsize = 420.0
>>> nthreads = 4
>>> autocorr = 1
>>> mu_max = 1.0
>>> seed = 42
>>> nmu_bins = 10
>>> np.random.seed(seed)
>>> X = np.random.uniform(0, boxsize, N)
>>> Y = np.random.uniform(0, boxsize, N)
>>> Z = np.random.uniform(0, boxsize, N)
>>> weights = np.ones_like(X)
>>> results = DDsmu(autocorr, nthreads, binfile, mu_max, nmu_bins,
...                 X, Y, Z, weights1=weights, weight_type='pair_product',␣
↪output_savg=True)
>>> for r in results[100:]: print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10.1f}"
...                               " {4:10d} {5:10.6f}".format(r['smin'], r['smax
↪'],
...                               r['savg'], r['mu_max'], r['npairs'], r[
↪'weightavg']))
...
 5.788530   8.249250   7.148213        0.1        230   1.000000
 5.788530   8.249250   7.157218        0.2        236   1.000000
 5.788530   8.249250   7.165338        0.3        208   1.000000
 5.788530   8.249250   7.079905        0.4        252   1.000000
 5.788530   8.249250   7.251661        0.5        184   1.000000
 5.788530   8.249250   7.118536        0.6        222   1.000000
 5.788530   8.249250   7.083466        0.7        238   1.000000
 5.788530   8.249250   7.198184        0.8        170   1.000000
 5.788530   8.249250   7.127409        0.9        208   1.000000
 5.788530   8.249250   6.973090        1.0        206   1.000000
 8.249250  11.756000  10.149183        0.1        592   1.000000
 8.249250  11.756000  10.213009        0.2        634   1.000000
 8.249250  11.756000  10.192220        0.3        532   1.000000
 8.249250  11.756000  10.246931        0.4        544   1.000000
 8.249250  11.756000  10.102675        0.5        530   1.000000
 8.249250  11.756000  10.276180        0.6        644   1.000000
 8.249250  11.756000  10.251264        0.7        666   1.000000
 8.249250  11.756000  10.138399        0.8        680   1.000000
 8.249250  11.756000  10.191916        0.9        566   1.000000
 8.249250  11.756000  10.243229        1.0        608   1.000000
11.756000  16.753600  14.552776        0.1       1734   1.000000
11.756000  16.753600  14.579991        0.2       1806   1.000000
11.756000  16.753600  14.599611        0.3       1802   1.000000
```

(continues on next page)

(continued from previous page)

```
11.756000  16.753600  14.471100        0.4       1820   1.000000
11.756000  16.753600  14.480192        0.5       1740   1.000000
11.756000  16.753600  14.493679        0.6       1746   1.000000
11.756000  16.753600  14.547713        0.7       1722   1.000000
11.756000  16.753600  14.465390        0.8       1750   1.000000
11.756000  16.753600  14.547465        0.9       1798   1.000000
11.756000  16.753600  14.440975        1.0       1828   1.000000
16.753600  23.875500  20.720406        0.1       5094   1.000000
16.753600  23.875500  20.735403        0.2       5004   1.000000
16.753600  23.875500  20.721069        0.3       5172   1.000000
16.753600  23.875500  20.723648        0.4       5014   1.000000
16.753600  23.875500  20.650621        0.5       5094   1.000000
16.753600  23.875500  20.688135        0.6       5076   1.000000
16.753600  23.875500  20.735691        0.7       4910   1.000000
16.753600  23.875500  20.714097        0.8       4864   1.000000
16.753600  23.875500  20.751836        0.9       4954   1.000000
16.753600  23.875500  20.721183        1.0       5070   1.000000
```

Submodules

Corrfunc.theory.DD module

Python wrapper around the C extension for the pair counter in `theory/DD/`. This wrapper is in *Corrfunc.theory.DD*

`Corrfunc.theory.DD.`**`DD`**(*autocorr*, *nthreads*, *binfile*, *X1*, *Y1*, *Z1*, *weights1=None*, *periodic=True*, *X2=None*, *Y2=None*, *Z2=None*, *weights2=None*, *verbose=False*, *boxsize=0.0*, *output_ravg=False*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*, *weight_type=None*)

Calculate the 3-D pair-counts corresponding to the real-space correlation function, $\xi(r)$.

If `weights` are provided, the resulting pair counts are weighted. The weighting scheme depends on `weight_type`.

Note: This module only returns pair counts and not the actual correlation function $\xi(r)$. See *Corrfunc.utils.convert_3d_counts_to_cf* for computing for computing $\xi(r)$ from the pair counts returned.

Parameters

- **autocorr** (*boolean, required*) – Boolean flag for auto/cross-correlation. If autocorr is set to 1, then the second set of particle positions are not required.
- **nthreads** (*integer*) – The number of OpenMP threads to use. Has no effect if OpenMP was not enabled during library compilation.
- **binfile** (*string or an list/array of floats*) – For string input: filename specifying the `r` bins for `DD`. The file should contain white-space separated values of (rmin, rmax) for each `r` wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

  For array-like input: A sequence of `r` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.

- **X1/Y1/Z1** (*array_like, real (float/double)*) – The array of X/Y/Z positions for the first set of points. Calculations are done in the precision of the supplied arrays.
- **weights1** (*array_like, real (float/double), optional*) – A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field. If only one of weights1 and weights2 is specified, the other will be set to uniform weights.
- **periodic** (*boolean*) – Boolean flag to indicate periodic boundary conditions.
- **X2/Y2/Z2** (*array-like, real (float/double)*) – Array of XYZ positions for the second set of points. *Must* be the same precision as the X1/Y1/Z1 arrays. Only required when `autocorr==0`.
- **weights2** (*array-like, real (float/double), optional*) – Same as weights1, but for the second set of positions
- **verbose** (*boolean (default false)*) – Boolean flag to control output of informational messages
- **boxsize** (*double*) – The side-length of the cube in the cosmological simulation. Present to facilitate exact calculations for periodic wrapping. If boxsize is not supplied, then the wrapping is done based on the maximum difference within each dimension of the X/Y/Z arrays.
- **output_ravg** (*boolean (default false)*) – Boolean flag to output the average `r` for each bin. Code will run slower if you set this flag.

  Note: If you are calculating in single-precision, `ravg` will suffer from numerical loss of precision and can not be trusted. If you need accurate `ravg` values, then pass in double precision arrays for the particle positions.

(xyz)bin_refine_factor: integer, default is (2,2,1); typically within [1-3] Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

max_cells_per_dim: integer, default is 100, typical values in [50-300] Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rmax` is too small relative to the boxsize (and increasing helps the runtime).

c_api_timer: boolean (default false) Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

isa: string (default `fastest`) Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

weight_type: string, optional The type of weighting to apply. One of ["pair_product", None]. Default: None.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [rmin, rmax, ravg, npairs, weightavg] for each radial bin specified in the `binfile`. If `output_ravg` is not set, then `ravg` will be set to 0.0 for all bins; similarly for `weightavg`. `npairs` contains

the number of pairs in that bin and can be used to compute the actual $\xi(r)$ by combining with (DR, RR) counts.

- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.theory.DD import DD
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../theory/tests/", "bins")
>>> N = 10000
>>> boxsize = 420.0
>>> nthreads = 4
>>> autocorr = 1
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(0, boxsize, N)
>>> Y = np.random.uniform(0, boxsize, N)
>>> Z = np.random.uniform(0, boxsize, N)
>>> weights = np.ones_like(X)
>>> results = DD(autocorr, nthreads, binfile, X, Y, Z, weights1=weights, weight_
↪type='pair_product', output_ravg=True)
>>> for r in results: print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10d} {4:10.6f}".
...                         format(r['rmin'], r['rmax'], r['ravg'],
...                         r['npairs'], r['weightavg']))
  0.167536   0.238755   0.000000          0   0.000000
  0.238755   0.340251   0.000000          0   0.000000
  0.340251   0.484892   0.000000          0   0.000000
  0.484892   0.691021   0.000000          0   0.000000
  0.691021   0.984777   0.945372          2   1.000000
  0.984777   1.403410   1.340525         10   1.000000
  1.403410   2.000000   1.732968         36   1.000000
  2.000000   2.850200   2.558878         54   1.000000
  2.850200   4.061840   3.564959        208   1.000000
  4.061840   5.788530   4.999278        674   1.000000
  5.788530   8.249250   7.126673       2154   1.000000
  8.249250  11.756000  10.201834       5996   1.000000
 11.756000  16.753600  14.517830      17746   1.000000
 16.753600  23.875500  20.716017      50252   1.000000
```

Corrfunc.theory.DDrppi module

Python wrapper around the C extension for the pair counter in `theory/DDrppi/`. This wrapper is in *Corrfunc.theory.DDrppi*

`Corrfunc.theory.DDrppi.`**`DDrppi`**(*autocorr*, *nthreads*, *pimax*, *binfile*, *X1*, *Y1*, *Z1*, *weights1=None*, *periodic=True*, *X2=None*, *Y2=None*, *Z2=None*, *weights2=None*, *verbose=False*, *boxsize=0.0*, *output_rpavg=False*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*, *weight_type=None*)

Calculate the 3-D pair-counts corresponding to the real-space correlation function, $\xi(r_p, \pi)$ or $\wp(r_p)$. Pairs which are separated by less than the `rp` bins (specified in `binfile`) in the X-Y plane, and less than `pimax` in the Z-dimension are counted.

If `weights` are provided, the resulting pair counts are weighted. The weighting scheme depends on `weight_type`.

Note: that this module only returns pair counts and not the actual correlation function $\xi(r_p, \pi)$ or $wp(r_p)$. See the utilities *Corrfunc.utils.convert_3d_counts_to_cf* and *Corrfunc.utils.convert_rp_pi_counts_to_wp* for computing $\xi(r_p, \pi)$ and $wp(r_p)$ respectively from the pair counts.

Parameters

- **autocorr** (*boolean, required*) – Boolean flag for auto/cross-correlation. If autocorr is set to 1, then the second set of particle positions are not required.
- **nthreads** (*integer*) – The number of OpenMP threads to use. Has no effect if OpenMP was not enabled during library compilation.
- **pimax** (*double*) – A double-precision value for the maximum separation along the Z-dimension.

  Distances along the :math:`\pi` direction are binned with unit depth. For instance, if `pimax=40`, then 40 bins will be created along the `pi` direction.

  Note: Only pairs with `0 <= dz < pimax` are counted (no equality).

binfile: string or an list/array of floats For string input: filename specifying the `rp` bins for `DDrppi`. The file should contain white-space separated values of (rpmin, rpmax) for each `rp` wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

For array-like input: A sequence of `rp` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.

X1/Y1/Z1: array-like, real (float/double) The array of X/Y/Z positions for the first set of points. Calculations are done in the precision of the supplied arrays.

weights1: array_like, real (float/double), optional A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field. If only one of weights1 and weights2 is specified, the other will be set to uniform weights.

X2/Y2/Z2: array-like, real (float/double) Array of XYZ positions for the second set of points. *Must* be the same precision as the X1/Y1/Z1 arrays. Only required when `autocorr==0`.

weights2: array-like, real (float/double), optional Same as weights1, but for the second set of positions

periodic: boolean Boolean flag to indicate periodic boundary conditions.

verbose: boolean (default false) Boolean flag to control output of informational messages

boxsize: double The side-length of the cube in the cosmological simulation. Present to facilitate exact calculations for periodic wrapping. If boxsize is not supplied, then the wrapping is done based on the maximum difference within each dimension of the X/Y/Z arrays.

output_rpavg: boolean (default false) Boolean flag to output the average `rp` for each bin. Code will run slower if you set this flag.

Note: If you are calculating in single-precision, `rpavg` will suffer from numerical loss of precision and can not be trusted. If you need accurate `rpavg` values, then pass in double precision arrays for the particle positions.

(xyz)bin_refine_factor: integer, default is (2,2,1); typically within [1-3] Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

max_cells_per_dim: integer, default is 100, typical values in [50-300] Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rpmax` is too small relative to the boxsize (and increasing helps the runtime).

c_api_timer: boolean (default false) Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

isa: string (default `fastest`) Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

weight_type: string, optional The type of weighting to apply. One of ["pair_product", None]. Default: None.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [rpmin, rpmax, rpavg, pimax, npairs, weightavg] for each radial bin specified in the `binfile`. If `output_rpavg` is not set, then `rpavg` will be set to 0.0 for all bins; similarly for `weightavg`. `npairs` contains the number of pairs in that bin and can be used to compute $\xi(r_p, \pi)$ by combining with (DR, RR) counts.
- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.theory.DDrppi import DDrppi
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../theory/tests/", "bins")
>>> N = 10000
>>> boxsize = 420.0
>>> nthreads = 4
>>> autocorr = 1
>>> pimax = 40.0
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(0, boxsize, N)
>>> Y = np.random.uniform(0, boxsize, N)
>>> Z = np.random.uniform(0, boxsize, N)
>>> weights = np.ones_like(X)
```

(continues on next page)

(continued from previous page)

```
>>> results = DDrppi(autocorr, nthreads, pimax, binfile,
...                  X, Y, Z, weights1=weights, weight_type='pair_product',␣
↪output_rpavg=True)
>>> for r in results[519:]: print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10.1f}"
...                               " {4:10d} {5:10.6f}".format(r['rmin'], r['rmax
↪'],
...                               r['rpavg'], r['pimax'], r['npairs'], r[
↪'weightavg']))
...
 11.756000  16.753600  14.379250       40.0       1150   1.000000
 16.753600  23.875500  20.449131        1.0       2604   1.000000
 16.753600  23.875500  20.604834        2.0       2370   1.000000
 16.753600  23.875500  20.523989        3.0       2428   1.000000
 16.753600  23.875500  20.475181        4.0       2462   1.000000
 16.753600  23.875500  20.458005        5.0       2532   1.000000
 16.753600  23.875500  20.537162        6.0       2522   1.000000
 16.753600  23.875500  20.443087        7.0       2422   1.000000
 16.753600  23.875500  20.474580        8.0       2360   1.000000
 16.753600  23.875500  20.420360        9.0       2512   1.000000
 16.753600  23.875500  20.478355       10.0       2472   1.000000
 16.753600  23.875500  20.485268       11.0       2406   1.000000
 16.753600  23.875500  20.372985       12.0       2420   1.000000
 16.753600  23.875500  20.647998       13.0       2378   1.000000
 16.753600  23.875500  20.556208       14.0       2420   1.000000
 16.753600  23.875500  20.527992       15.0       2462   1.000000
 16.753600  23.875500  20.581017       16.0       2380   1.000000
 16.753600  23.875500  20.491819       17.0       2346   1.000000
 16.753600  23.875500  20.534440       18.0       2496   1.000000
 16.753600  23.875500  20.529129       19.0       2512   1.000000
 16.753600  23.875500  20.501946       20.0       2500   1.000000
 16.753600  23.875500  20.513349       21.0       2544   1.000000
 16.753600  23.875500  20.471915       22.0       2430   1.000000
 16.753600  23.875500  20.450651       23.0       2354   1.000000
 16.753600  23.875500  20.550753       24.0       2460   1.000000
 16.753600  23.875500  20.540262       25.0       2490   1.000000
 16.753600  23.875500  20.559572       26.0       2350   1.000000
 16.753600  23.875500  20.534245       27.0       2382   1.000000
 16.753600  23.875500  20.511302       28.0       2508   1.000000
 16.753600  23.875500  20.491632       29.0       2456   1.000000
 16.753600  23.875500  20.592493       30.0       2386   1.000000
 16.753600  23.875500  20.506234       31.0       2484   1.000000
 16.753600  23.875500  20.482109       32.0       2538   1.000000
 16.753600  23.875500  20.518463       33.0       2544   1.000000
 16.753600  23.875500  20.482515       34.0       2534   1.000000
 16.753600  23.875500  20.503124       35.0       2382   1.000000
 16.753600  23.875500  20.471307       36.0       2356   1.000000
 16.753600  23.875500  20.384231       37.0       2554   1.000000
 16.753600  23.875500  20.454012       38.0       2458   1.000000
 16.753600  23.875500  20.585543       39.0       2394   1.000000
 16.753600  23.875500  20.504965       40.0       2500   1.000000
```

Corrfunc.theory.DDsmu module

Python wrapper around the C extension for the pair counter in `theory/DDsmu/`. This wrapper is in *Corrfunc.theory.DDsmu*

Corrfunc.theory.DDsmu.**DDsmu**(*autocorr*, *nthreads*, *binfile*, *mu_max*, *nmu_bins*, *X1*, *Y1*, *Z1*, *weights1=None*, *periodic=True*, *X2=None*, *Y2=None*, *Z2=None*, *weights2=None*, *verbose=False*, *boxsize=0.0*, *output_savg=False*, *fast_divide_and_NR_steps=0*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*, *weight_type=None*)

Calculate the 2-D pair-counts corresponding to the redshift-space correlation function, $\xi(s, \mu)$ Pairs which are separated by less than the `s` bins (specified in `binfile`) in 3-D, and less than `s*mu_max` in the Z-dimension are counted.

If `weights` are provided, the resulting pair counts are weighted. The weighting scheme depends on `weight_type`.

Note: This module only returns pair counts and not the actual correlation function $\xi(s, \mu)$. See the utilities *Corrfunc.utils.convert_3d_counts_to_cf* for computing $\xi(s, \mu)$ from the pair counts.

New in version 2.1.0.

Parameters

- **autocorr** (*boolean, required*) – Boolean flag for auto/cross-correlation. If autocorr is set to 1, then the second set of particle positions are not required.
- **nthreads** (*integer*) – The number of OpenMP threads to use. Has no effect if OpenMP was not enabled during library compilation.
- **binfile** (*string or an list/array of floats*) – For string input: filename specifying the `s` bins for `DDsmu_mocks`. The file should contain white-space separated values of (smin, smax) specifying each `s` bin wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

  For array-like input: A sequence of `s` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.
- **mu_max** (*double. Must be in range (0.0, 1.0]*) – A double-precision value for the maximum cosine of the angular separation from the line of sight (LOS). Here, LOS is taken to be along the Z direction.

  Note: Only pairs with $0 <= \cos(\theta_{LOS}) < \mu_{max}$ are counted (no equality).
- **nmu_bins** (*int*) – The number of linear `mu` bins, with the bins ranging from from (0, μ_{max})
- **X1/Y1/Z1** (*array-like, real (float/double)*) – The array of X/Y/Z positions for the first set of points. Calculations are done in the precision of the supplied arrays.
- **weights1** (*array-like, real (float/double), shape (n_particles,) or (n_weights_per_particle,n_particles), optional*) – Weights for computing a weighted pair count.
- **weight_type** (*str, optional*) – The type of pair weighting to apply. Options: "pair_product", None; Default: None.
- **periodic** (*boolean*) – Boolean flag to indicate periodic boundary conditions.
- **X2/Y2/Z2** (*array-like, real (float/double)*) – Array of XYZ positions for the second set of points. *Must* be the same precision as the X1/Y1/Z1 arrays. Only required when `autocorr==0`.

- **weights2** (*array-like, real (float/double), shape (n_particles,) or (n_weights_per_particle,n_particles), optional*) – Weights for computing a weighted pair count.
- **verbose** (*boolean (default false)*) – Boolean flag to control output of informational messages
- **boxsize** (*double*) – The side-length of the cube in the cosmological simulation. Present to facilitate exact calculations for periodic wrapping. If boxsize is not supplied, then the wrapping is done based on the maximum difference within each dimension of the X/Y/Z arrays.
- **output_savg** (*boolean (default false)*) – Boolean flag to output the average `s` for each bin. Code will run slower if you set this flag. Also, note, if you are calculating in single-precision, `s` will suffer from numerical loss of precision and can not be trusted. If you need accurate `s` values, then pass in double precision arrays for the particle positions.
- **fast_divide_and_NR_steps** (*integer (default 0)*) – Replaces the division in `AVX` implementation with an approximate reciprocal, followed by `fast_divide_and_NR_steps` of Newton-Raphson. Can improve runtime by ~15-20% on older computers. Value of 0 uses the standard division operation.
- **(xyz)bin_refine_factor** (*integer (default (2,2,1) typical values in [1-3])*) – Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.
- **max_cells_per_dim** (*integer (default 100, typical values in [50-300])*) – Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rmax` is too small relative to the boxsize (and increasing helps the runtime).
- **c_api_timer** (*boolean (default false)*) – Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.
- **isa** (*integer (default -1)*) – Controls the runtime dispatch for the instruction set to use. Possible options are: [-1, AVX, SSE42, FALLBACK]

  Setting isa to -1 will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `AVX` and `AVX` is not available on the computer, then the code will revert to using `FALLBACK` (even though `SSE42` might be available).

  Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the integer values correspond to the `enum` for the instruction set defined in `utils/defs.h`.

Returns

- **results** (*A python list*) – A python list containing `nmu_bins` of [smin, smax, savg, mu_max, npairs, weightavg] for each spatial bin specified in the `binfile`. There will be a total of `nmu_bins` ranging from [0, `mu_max`) *per* spatial bin. If `output_savg` is not set, then `savg` will be set to 0.0 for all bins; similarly for `weight_avg`. `npairs` contains the number of pairs in that bin.
- **time** (if `c_api_timer` is set, then the return value contains the time spent) – in the API; otherwise time is set to 0.0

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.theory.DDsmu import DDsmu
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../theory/tests/", "bins")
>>> N = 10000
>>> boxsize = 420.0
>>> nthreads = 4
>>> autocorr = 1
>>> mu_max = 1.0
>>> seed = 42
>>> nmu_bins = 10
>>> np.random.seed(seed)
>>> X = np.random.uniform(0, boxsize, N)
>>> Y = np.random.uniform(0, boxsize, N)
>>> Z = np.random.uniform(0, boxsize, N)
>>> weights = np.ones_like(X)
>>> results = DDsmu(autocorr, nthreads, binfile, mu_max, nmu_bins,
...                 X, Y, Z, weights1=weights, weight_type='pair_product',␣
↪output_savg=True)
>>> for r in results[100:]: print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10.1f}"
...                               " {4:10d} {5:10.6f}".format(r['smin'], r['smax
↪'],
...                               r['savg'], r['mu_max'], r['npairs'], r[
↪'weightavg']))
...
 5.788530   8.249250   7.148213        0.1        230   1.000000
 5.788530   8.249250   7.157218        0.2        236   1.000000
 5.788530   8.249250   7.165338        0.3        208   1.000000
 5.788530   8.249250   7.079905        0.4        252   1.000000
 5.788530   8.249250   7.251661        0.5        184   1.000000
 5.788530   8.249250   7.118536        0.6        222   1.000000
 5.788530   8.249250   7.083466        0.7        238   1.000000
 5.788530   8.249250   7.198184        0.8        170   1.000000
 5.788530   8.249250   7.127409        0.9        208   1.000000
 5.788530   8.249250   6.973090        1.0        206   1.000000
 8.249250  11.756000  10.149183        0.1        592   1.000000
 8.249250  11.756000  10.213009        0.2        634   1.000000
 8.249250  11.756000  10.192220        0.3        532   1.000000
 8.249250  11.756000  10.246931        0.4        544   1.000000
 8.249250  11.756000  10.102675        0.5        530   1.000000
 8.249250  11.756000  10.276180        0.6        644   1.000000
 8.249250  11.756000  10.251264        0.7        666   1.000000
 8.249250  11.756000  10.138399        0.8        680   1.000000
 8.249250  11.756000  10.191916        0.9        566   1.000000
 8.249250  11.756000  10.243229        1.0        608   1.000000
11.756000  16.753600  14.552776        0.1       1734   1.000000
11.756000  16.753600  14.579991        0.2       1806   1.000000
11.756000  16.753600  14.599611        0.3       1802   1.000000
11.756000  16.753600  14.471100        0.4       1820   1.000000
11.756000  16.753600  14.480192        0.5       1740   1.000000
11.756000  16.753600  14.493679        0.6       1746   1.000000
11.756000  16.753600  14.547713        0.7       1722   1.000000
```

(continues on next page)

(continued from previous page)

```
11.756000  16.753600  14.465390        0.8       1750   1.000000
11.756000  16.753600  14.547465        0.9       1798   1.000000
11.756000  16.753600  14.440975        1.0       1828   1.000000
16.753600  23.875500  20.720406        0.1       5094   1.000000
16.753600  23.875500  20.735403        0.2       5004   1.000000
16.753600  23.875500  20.721069        0.3       5172   1.000000
16.753600  23.875500  20.723648        0.4       5014   1.000000
16.753600  23.875500  20.650621        0.5       5094   1.000000
16.753600  23.875500  20.688135        0.6       5076   1.000000
16.753600  23.875500  20.735691        0.7       4910   1.000000
16.753600  23.875500  20.714097        0.8       4864   1.000000
16.753600  23.875500  20.751836        0.9       4954   1.000000
16.753600  23.875500  20.721183        1.0       5070   1.000000
```

Corrfunc.theory.vpf module

Python wrapper around the C extension for the counts-in-cells for 3-D real space. Corresponding C codes are in `theory/vpf` while the python wrapper is in *Corrfunc.theory.vpf*.

`Corrfunc.theory.vpf.`**`vpf`**(*rmax*, *nbins*, *nspheres*, *numpN*, *seed*, *X*, *Y*, *Z*, *verbose=False*, *periodic=True*, *boxsize=0.0*, *xbin_refine_factor=1*, *ybin_refine_factor=1*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*)

Function to compute the counts-in-cells on 3-D real-space points.

Returns a numpy structured array containing the probability of a sphere of radius up to `rmax` containing [0, numpN-1] galaxies.

Parameters

- **`rmax`** (*`double`*) – Maximum radius of the sphere to place on the particles
- **`nbins`** (*`integer`*) – Number of bins in the counts-in-cells. Radius of first shell is rmax/nbins
- **`nspheres`** (*`integer (>= 0)`*) – Number of random spheres to place within the particle distribution. For a small number of spheres, the error is larger in the measured pN's.
- **`numpN`** (*`integer (>= 1)`*) – Governs how many unique pN's are to returned. If `numpN` is set to 1, then only the vpf (p0) is returned. For `numpN=2`, p0 and p1 are returned.

  More explicitly, the columns in the results look like the following:

numpN	Columns in output
1	p0
2	p0 p1
3	p0 p1 p2
4	p0 p1 p2 p3

  and so on...

  Note: `p0` is the vpf

seed: unsigned integer Random number seed for the underlying GSL random number generator. Used to draw centers of the spheres.

X/Y/Z: arraytype, real (float/double) Particle positions in the 3 axes. Must be within [0, boxsize] and specified in the same units as `rp_bins` and boxsize. All 3 arrays must be of the same floating-point type.

Calculations will be done in the same precision as these arrays, i.e., calculations will be in floating point if XYZ are single precision arrays (C float type); or in double-precision if XYZ are double precision arrays (C double type).

verbose: boolean (default false) Boolean flag to control output of informational messages

periodic: boolean Boolean flag to indicate periodic boundary conditions.

boxsize: double The side-length of the cube in the cosmological simulation. Present to facilitate exact calculations for periodic wrapping. If boxsize is not supplied, then the wrapping is done based on the maximum difference within each dimension of the X/Y/Z arrays.

(xyz)bin_refine_factor: integer, default is (1,1,1); typically within [1-3] Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

Note: Since the counts in spheres calculation is symmetric in all 3 dimensions, the defaults are different from the clustering routines.

max_cells_per_dim: integer, default is 100, typical values in [50-300] Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rmax` is too small relative to the boxsize (and increasing helps the runtime).

c_api_timer: boolean (default false) Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

isa: string (default `fastest`) Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

Returns

results – A numpy structured array containing [rmax, pN[numpN]] with `nbins` elements. Each row contains the maximum radius of the sphere and the `numpN` elements in the `pN` array. Each element of this array contains the probability that a sphere of radius `rmax` contains *exactly* `N` galaxies. For example, pN[0] (p0, the void probibility function) is the probability that a sphere of radius `rmax` contains 0 galaxies.

if `c_api_timer` is set, then the return value is a tuple containing (results, api_time). `api_time` measures only the time spent within the C library and ignores all python overhead.

Return type Numpy structured array

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from Corrfunc.theory.vpf import vpf
>>> rmax = 10.0
>>> nbins = 10
```

(continues on next page)

(continued from previous page)

```
>>> nspheres = 10000
>>> numpN = 5
>>> seed = -1
>>> N = 100000
>>> boxsize = 420.0
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(0, boxsize, N)
>>> Y = np.random.uniform(0, boxsize, N)
>>> Z = np.random.uniform(0, boxsize, N)
>>> results = vpf(rmax, nbins, nspheres, numpN, seed, X, Y, Z)
>>> for r in results:
...     print("{0:10.1f} ".format(r[0]), end="")
...
...     for pn in r[1]:
...         print("{0:10.3f} ".format(pn), end="")
...
...     print("")
1.0       0.995      0.005      0.000      0.000      0.000
2.0       0.956      0.044      0.001      0.000      0.000
3.0       0.858      0.130      0.012      0.001      0.000
4.0       0.695      0.252      0.047      0.005      0.001
5.0       0.493      0.347      0.127      0.028      0.005
6.0       0.295      0.362      0.219      0.091      0.026
7.0       0.141      0.285      0.265      0.179      0.085
8.0       0.056      0.159      0.228      0.229      0.161
9.0       0.019      0.066      0.135      0.192      0.192
10.0      0.003      0.019      0.054      0.106      0.150
```

Corrfunc.theory.wp module

Python wrapper around the C extension for the theoretical projected auto-correlation function, wp(rp), in `theory/wp`. This python wrapper is in *Corrfunc.theory.wp*.

`Corrfunc.theory.wp.`**`wp`**(*boxsize*, *pimax*, *nthreads*, *binfile*, *X*, *Y*, *Z*, *weights=None*, *weight_type=None*, *verbose=False*, *output_rpavg=False*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *c_cell_timer=False*, *isa=u'fastest'*)

Function to compute the projected correlation function in a periodic cosmological box. Pairs which are separated by less than the `rp` bins (specified in `binfile`) in the X-Y plane, and less than `pimax` in the Z-dimension are counted.

If `weights` are provided, the resulting correlation function is weighted. The weighting scheme depends on `weight_type`.

Note: Pairs are double-counted. And if `rpmin` is set to 0.0, then all the self-pairs (i'th particle with itself) are added to the first bin => minimum number of pairs in the first bin is the total number of particles.

Parameters

- **`boxsize`** (*`double`*) – A double-precision value for the boxsize of the simulation in same units as the particle positions and the `rp` bins.

- **pimax** (*double*) – A double-precision value for the maximum separation along the Z-dimension.

  Note: Only pairs with `0 <= dz < pimax` are counted (no equality).

nthreads: integer Number of threads to use.

binfile: string or an list/array of floats For string input: filename specifying the `rp` bins for `wp`. The file should contain white-space separated values of (rpmin, rpmax) for each `rp` wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

For array-like input: A sequence of `rp` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.

X/Y/Z: arraytype, real (float/double) Particle positions in the 3 axes. Must be within [0, boxsize] and specified in the same units as `rp_bins` and boxsize. All 3 arrays must be of the same floating-point type.

Calculations will be done in the same precision as these arrays, i.e., calculations will be in floating point if XYZ are single precision arrays (C float type); or in double-precision if XYZ are double precision arrays (C double type).

weights: array_like, real (float/double), optional A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field.

verbose: boolean (default false) Boolean flag to control output of informational messages

output_rpavg: boolean (default false) Boolean flag to output the average `rp` for each bin. Code will run slower if you set this flag.

Note: If you are calculating in single-precision, `rpavg` will suffer from numerical loss of precision and can not be trusted. If you need accurate `rpavg` values, then pass in double precision arrays for the particle positions.

(xyz)bin_refine_factor: integer, default is (2,2,1); typically within [1-3] Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

max_cells_per_dim: integer, default is 100, typical values in [50-300] Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rpmax` is too small relative to the boxsize (and increasing helps the runtime).

c_api_timer: boolean (default false) Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

c_cell_timer [boolean (default false)] Boolean flag to measure actual time spent **per cell-pair** within the C libraries. A very detailed timer that stores information about the number of particles in each cell, the thread id that processed that cell-pair and the amount of time in nano-seconds taken to process that cell pair. This timer can be used to study the instruction set efficiency, and load-balancing of the code.

isa: string (default `fastest`) Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

weight_type: string, optional The type of weighting to apply. One of ["pair_product", None]. Default: None.

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [rpmin, rpmax, rpavg, wp, npairs, weightavg] for each radial specified in the `binfile`. If `output_rpavg` is not set then `rpavg` will be set to 0.0 for all bins; similarly for `weightavg`. `wp` contains the projected correlation function while `npairs` contains the number of unique pairs in that bin. If using weights, `wp` will be weighted while `npairs` will not be.
- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.
- **cell_time** (*list, optional*) – Only returned if `c_cell_timer` is set. Contains detailed stats about each cell-pair visited during pair-counting, viz., number of particles in each of the cells in the pair, 1-D cell-indices for each cell in the pair, time (in nano-seconds) to process the pair and the thread-id for the thread that processed that cell-pair.

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.theory.wp import wp
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../theory/tests/", "bins")
>>> N = 10000
>>> boxsize = 420.0
>>> pimax = 40.0
>>> nthreads = 4
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(0, boxsize, N)
>>> Y = np.random.uniform(0, boxsize, N)
>>> Z = np.random.uniform(0, boxsize, N)
>>> results = wp(boxsize, pimax, nthreads, binfile, X, Y, Z, weights=np.ones_
↪like(X), weight_type='pair_product')
>>> for r in results:
...     print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10.6f} {4:10d} {5:10.6f}".
...           format(r['rmin'], r['rmax'],
...           r['rpavg'], r['wp'], r['npairs'], r['weightavg']))
...
  0.167536   0.238755   0.000000  66.717143         18   1.000000
  0.238755   0.340251   0.000000 -15.786045         16   1.000000
  0.340251   0.484892   0.000000   2.998470         42   1.000000
  0.484892   0.691021   0.000000 -15.779885         66   1.000000
  0.691021   0.984777   0.000000 -11.966728        142   1.000000
  0.984777   1.403410   0.000000  -9.699906        298   1.000000
  1.403410   2.000000   0.000000 -11.698771        588   1.000000
  2.000000   2.850200   0.000000   3.848375       1466   1.000000
  2.850200   4.061840   0.000000  -0.921452       2808   1.000000
  4.061840   5.788530   0.000000   0.454851       5802   1.000000
  5.788530   8.249250   0.000000   1.428344      11926   1.000000
  8.249250  11.756000   0.000000  -1.067885      23478   1.000000
 11.756000  16.753600   0.000000  -0.553319      47994   1.000000
 16.753600  23.875500   0.000000  -0.086433      98042   1.000000
```

Corrfunc.theory.wp.**find_fastest_wp_bin_refs**(*boxsize*, *pimax*, *nthreads*, *binfile*, *X*, *Y*, *Z*, *verbose=False*, *output_rpavg=False*, *max_cells_per_dim=100*, *isa=u'fastest'*, *maxbinref=3*, *nrepeats=3*, *return_runtimes=False*)

Finds the combination of `bin refine factors` that produces the fastest computation for the given dataset and `rp` limits.

Parameters

- **boxsize** (*double*) – A double-precision value for the boxsize of the simulation in same units as the particle positions and the `rp` bins.
- **pimax** (*double*) – A double-precision value for the maximum separation along the Z-dimension.

  Note: Only pairs with `0 <= dz < pimax` are counted (no equality).

nthreads: integer Number of threads to use.

binfile: string or an list/array of floats For string input: filename specifying the `rp` bins for `wp`. The file should contain white-space separated values of (rpmin, rpmax) for each `rp` wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

For array-like input: A sequence of `rp` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.

X/Y/Z: arraytype, real (float/double) Particle positions in the 3 axes. Must be within [0, boxsize] and specified in the same units as `rp_bins` and boxsize. All 3 arrays must be of the same floating-point type.

Calculations will be done in the same precision as these arrays, i.e., calculations will be in floating point if XYZ are single precision arrays (C float type); or in double-precision if XYZ are double precision arrays (C double type).

verbose: boolean (default false) Boolean flag to control output of informational messages

output_rpavg: boolean (default false) Boolean flag to output the average `rp` for each bin. Code will run slower if you set this flag.

Note: If you are calculating in single-precision, `rpavg` will suffer from numerical loss of precision and can not be trusted. If you need accurate `rpavg` values, then pass in double precision arrays for the particle positions.

max_cells_per_dim: integer, default is 100, typical values in [50-300] Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rpmax` is too small relative to the boxsize (and increasing helps the runtime).

isa: string (default `fastest`) Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

maxbinref: integer (default 3) The maximum `bin refine factor` to use along each dimension. From experience, values larger than 3 do not improve `wp` runtime.

Runtime of module scales as `maxbinref^3`, so change the value of `maxbinref` with caution.

nrepeats: integer (default 3) Number of times to repeat the timing for an individual run. Accounts for the dispersion in runtimes on computers with multiple user processes.

return_runtimes: boolean (default `false`) If set, also returns the array of runtimes.

Returns

- **(nx, ny, nz)** (*tuple of integers*) – The combination of `bin refine factors` along each dimension that produces the fastest code.
- **runtimes** (*numpy structured array*) – if `return_runtimes` is set, then the return value is a tuple containing ((nx, ny, nz), runtimes). `runtimes` is a `numpy` structured array containing the fields, [`nx`, `ny`, `nz`, `avg_runtime`, `sigma_time`]. Here, `avg_runtime` is the average time, measured over `nrepeats` invocations, spent in the python extension. `sigma_time` is the dispersion of the run times across those `nrepeats` invocations.

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.io import read_catalog
>>> from Corrfunc.theory.wp import find_fastest_wp_bin_refs
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../theory/tests/", "bins")
>>> X, Y, Z = read_catalog(return_dtype=np.float32)
>>> boxsize = 420.0
>>> pimax = 40.0
>>> nthreads = 4
>>> verbose = 1
>>> best, _ = find_fastest_wp_bin_refs(boxsize, pimax, nthreads, binfile,
...                                    X, Y, Z, maxbinref=2, nrepeats=3,
...                                    verbose=verbose,
...                                    return_runtimes=True)
>>> print(best)
(2, 2, 1)
```

Note: Since the result might change depending on the computer, doctest is skipped for this function.

Corrfunc.theory.xi module

Python wrapper around the C extension for the theoretical 3-D real-space correlation function, $\xi(r)$. Corresponding C routines are in `theory/xi/`, python interface is *`Corrfunc.theory.xi`*.

`Corrfunc.theory.xi.`**`xi`**(*boxsize*, *nthreads*, *binfile*, *X*, *Y*, *Z*, *weights=None*, *weight_type=None*, *verbose=False*, *output_ravg=False*, *xbin_refine_factor=2*, *ybin_refine_factor=2*, *zbin_refine_factor=1*, *max_cells_per_dim=100*, *c_api_timer=False*, *isa=u'fastest'*)

Function to compute the projected correlation function in a periodic cosmological box. Pairs which are separated by less than the `r` bins (specified in `binfile`) in 3-D real space.

If `weights` are provided, the resulting correlation function is weighted. The weighting scheme depends on `weight_type`.

Note: Pairs are double-counted. And if `rmin` is set to 0.0, then all the self-pairs (i'th particle with itself) are added to the first bin => minimum number of pairs in the first bin is the total number of particles.

Parameters

- **`boxsize`** (*`double`*) – A double-precision value for the boxsize of the simulation in same units as the particle positions and the `r` bins.
- **`nthreads`** (*`integer`*) – Number of threads to use.
- **`binfile`** (*`string or an list/array of floats`*) – For string input: filename specifying the `r` bins for `xi`. The file should contain white-space separated values of (rmin, rmax) for each `r` wanted. The bins need to be contiguous and sorted in increasing order (smallest bins come first).

  For array-like input: A sequence of `r` values that provides the bin-edges. For example, `np.logspace(np.log10(0.1), np.log10(10.0), 15)` is a valid input specifying **14** (logarithmic) bins between 0.1 and 10.0. This array does not need to be sorted.
- **`X/Y/Z`** (*`arraytype, real (float/double)`*) – Particle positions in the 3 axes. Must be within [0, boxsize] and specified in the same units as `rp_bins` and boxsize. All 3 arrays must be of the same floating-point type.

  Calculations will be done in the same precision as these arrays, i.e., calculations will be in floating point if XYZ are single precision arrays (C float type); or in double-precision if XYZ are double precision arrays (C double type).
- **`weights`** (*`array_like, real (float/double), optional`*) – A scalar, or an array of weights of shape (n_weights, n_positions) or (n_positions,). *weight_type* specifies how these weights are used; results are returned in the *weightavg* field.
- **`verbose`** (*`boolean (default false)`*) – Boolean flag to control output of informational messages
- **`output_ravg`** (*`boolean (default false)`*) – Boolean flag to output the average `r` for each bin. Code will run slower if you set this flag.

  Note: If you are calculating in single-precision, `rpavg` will suffer from numerical loss of precision and can not be trusted. If you need accurate `rpavg` values, then pass in double precision arrays for the particle positions.

(xyz)bin_refine_factor: integer, default is (2,2,1); typically within [1-3] Controls the refinement on the cell sizes. Can have up to a 20% impact on runtime.

max_cells_per_dim: integer, default is 100, typical values in [50-300] Controls the maximum number of cells per dimension. Total number of cells can be up to (max_cells_per_dim)^3. Only increase if `rmax` is too small relative to the boxsize (and increasing helps the runtime).

c_api_timer: boolean (default false) Boolean flag to measure actual time spent in the C libraries. Here to allow for benchmarking and scaling studies.

isa: string (default `fastest`) Controls the runtime dispatch for the instruction set to use. Possible options are: [`fastest`, `avx`, `sse42`, `fallback`]

Setting isa to `fastest` will pick the fastest available instruction set on the current computer. However, if you set `isa` to, say, `avx` and `avx` is not available on the computer, then the code will revert to using `fallback` (even though `sse42` might be available).

Unless you are benchmarking the different instruction sets, you should always leave `isa` to the default value. And if you *are* benchmarking, then the string supplied here gets translated into an `enum` for the instruction set defined in `utils/defs.h`.

weight_type: string, optional, Default: None. The type of weighting to apply. One of ["pair_product", None].

Returns

- **results** (*Numpy structured array*) – A numpy structured array containing [rmin, rmax, ravg, xi, npairs, weightavg] for each radial specified in the `binfile`. If `output_ravg` is not set then `ravg` will be set to 0.0 for all bins; similarly for `weightavg`. `xi` contains the correlation function while `npairs` contains the number of pairs in that bin. If using weights, `xi` will be weighted while `npairs` will not be.
- **api_time** (*float, optional*) – Only returned if `c_api_timer` is set. `api_time` measures only the time spent within the C library and ignores all python overhead.

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.theory.xi import xi
>>> binfile = pjoin(dirname(abspath(Corrfunc.__file__)),
...                 "../theory/tests/", "bins")
>>> N = 100000
>>> boxsize = 420.0
>>> nthreads = 4
>>> seed = 42
>>> np.random.seed(seed)
>>> X = np.random.uniform(0, boxsize, N)
>>> Y = np.random.uniform(0, boxsize, N)
>>> Z = np.random.uniform(0, boxsize, N)
>>> weights = np.ones_like(X)
>>> results = xi(boxsize, nthreads, binfile, X, Y, Z, weights=weights, weight_
↪type='pair_product', output_ravg=True)
>>> for r in results: print("{0:10.6f} {1:10.6f} {2:10.6f} {3:10.6f} {4:10d}
↪{5:10.6f}"
...                      .format(r['rmin'], r['rmax'],
...                      r['ravg'], r['xi'], r['npairs'], r['weightavg']))
...
  0.167536   0.238755   0.226592  -0.205733          4   1.000000
  0.238755   0.340251   0.289277  -0.176729         12   1.000000
  0.340251   0.484892   0.426819  -0.051829         40   1.000000
  0.484892   0.691021   0.596187  -0.131853        106   1.000000
  0.691021   0.984777   0.850100  -0.049207        336   1.000000
  0.984777   1.403410   1.225112   0.028543       1052   1.000000
  1.403410   2.000000   1.737153   0.011403       2994   1.000000
  2.000000   2.850200   2.474588   0.005405       8614   1.000000
  2.850200   4.061840   3.532018  -0.014098      24448   1.000000
  4.061840   5.788530   5.022241  -0.010784      70996   1.000000
  5.788530   8.249250   7.160648  -0.001588     207392   1.000000
  8.249250  11.756000  10.207213  -0.000323     601002   1.000000
 11.756000  16.753600  14.541171   0.000007    1740084   1.000000
 16.753600  23.875500  20.728773  -0.001595    5028058   1.000000
```

Submodules

Corrfunc.call_correlation_functions module

Corrfunc.call_correlation_functions_mocks module

Example python code to call the mocks clustering functions from python. This script calls the python extensions directly; however the recommended use is via the wrappers provided in *Corrfunc.mocks*.

Corrfunc.call_correlation_functions_mocks.**main**()

Corrfunc.io module

Routines to read galaxy catalogs from disk.

Corrfunc.io.**read_fastfood_catalog**(*filename*, *return_dtype=None*, *need_header=None*)

Read a galaxy catalog from a fast-food binary file.

Parameters

- **filename** (*string*) – Filename containing the galaxy positions
- **return_dtype** (numpy dtype for returned arrays. Default numpy.float) – Specifies the datatype for the returned arrays. Must be in {np.float, np.float32}
- **need_header** (*boolean, default None.*) – Returns the header found in the fast-food file in addition to the X/Y/Z arrays.

Returns

X, Y, Z – Returns the triplet of X/Y/Z positions as separate numpy arrays.

If need_header is set, then the header is also returned

Return type numpy arrays

Example

```
>>> import numpy as np
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.io import read_fastfood_catalog
>>> filename = pjoin(dirname(abspath(Corrfunc.__file__)),
...                  "../theory/tests/data/",
...                  "gals_Mr19.ff")
>>> X, Y, Z = read_fastfood_catalog(filename)
>>> N = 20
>>> for x,y,z in zip(X[0:N], Y[0:N], Z[0:]):
...     print("{0:10.5f} {1:10.5f} {2:10.5f}".format(x, y, z))
...
419.94550    1.96340    0.01610
419.88272    1.79736    0.11960
0.32880   10.63620    4.16550
0.15314   10.68723    4.06529
0.46400    8.91150    6.97090
6.30690    9.77090    8.61080
5.87160    9.65870    9.29810
8.06210    0.42350    4.89410
```

(continues on next page)

(continued from previous page)

```
11.92830    4.38660    4.54410
11.95543    4.32622    4.51485
11.65676    4.34665    4.53181
11.75739    4.26262    4.31666
11.81329    4.27530    4.49183
11.80406    4.54737    4.26824
12.61570    4.14470    3.70140
13.23640    4.34750    5.26450
13.19833    4.33196    5.29435
13.21249    4.35695    5.37418
13.06805    4.24275    5.35126
13.19693    4.37618    5.28772
```

Corrfunc.io.**read_ascii_catalog**(*filename*, *return_dtype=None*)

Read a galaxy catalog from an ascii file.

Parameters

- **filename** (*string*) – Filename containing the galaxy positions
- **return_dtype** (numpy dtype for returned arrays. Default `numpy.float`) – Specifies the datatype for the returned arrays. Must be in {np.float, np.float32}

Returns **X, Y, Z** – Returns the triplet of X/Y/Z positions as separate numpy arrays.

Return type numpy arrays

Example

```
>>> from __future__ import print_function
>>> from os.path import dirname, abspath, join as pjoin
>>> import Corrfunc
>>> from Corrfunc.io import read_ascii_catalog
>>> filename = pjoin(dirname(abspath(Corrfunc.__file__)),
...                  "../mocks/tests/data/", "Mr19_mock_northonly.rdcz.dat")
>>> ra, dec, cz = read_ascii_catalog(filename)
>>> N = 20
>>> for r,d,c in zip(ra[0:N], dec[0:N], cz[0:]):
...     print("{0:10.5f} {1:10.5f} {2:10.5f}".format(r, d, c))
...
178.45087   67.01112 19905.28514
178.83495   67.72519 19824.02285
179.50132   67.67628 19831.21553
182.75497   67.13004 19659.79825
186.29853   68.64099 20030.64412
186.32346   68.65879 19763.38137
187.36173   68.15151 19942.66996
187.20613   68.56189 19996.36607
185.56358   67.97724 19729.32308
183.27930   67.11318 19609.71345
183.86498   67.82823 19500.44130
184.07771   67.43429 19440.53790
185.13370   67.15382 19390.60304
189.15907   68.28252 19858.85853
190.12209   68.55062 20044.29744
193.65245   68.36878 19445.62469
```

(continues on next page)

(continued from previous page)

```
194.93514   68.34870 19158.93155
180.36897   67.50058 18671.40780
179.63278   67.51318 18657.59191
180.75742   67.95530 18586.88913
```

Corrfunc.io.**read_catalog**(*filebase=None*, *return_dtype=<Mock id='140313965584656'>*)

Reads a galaxy/randoms catalog and returns 3 XYZ arrays.

Parameters

- **filebase** (*string (optional)*) – The fully qualified path to the file. If omitted, reads the theory galaxy catalog under ../theory/tests/data/
- **return_dtype** (numpy dtype for returned arrays. Default numpy.float) – Specifies the datatype for the returned arrays. Must be in {np.float, np.float32}

Returns

- x y z - Unpacked numpy arrays compatible with the installed
- version of Corrfunc.

Note: If the filename is omitted, then first the fast-food file is searched for, and then the ascii file. End-users should always supply the full filename.

Corrfunc.tests module

Corrfunc.tests.**tests**()

Wrapper to run the two scripts that should have been installed with the Corrfunc package.

If the two scripts (one for theory extensions, one for mocks extensions) run successfully, then the package is working correctly.

Corrfunc.utils module

A set of utility routines

Corrfunc.utils.**convert_3d_counts_to_cf**(*ND1*, *ND2*, *NR1*, *NR2*, *D1D2*, *D1R2*, *D2R1*, *R1R2*, *estimator='LS'*)

Converts raw pair counts to a correlation function.

Parameters

- **ND1** (*integer*) – Number of points in the first dataset
- **ND2** (*integer*) – Number of points in the second dataset
- **NR1** (*integer*) – Number of points in the randoms for first dataset
- **NR2** (*integer*) – Number of points in the randoms for second dataset
- **D1D2** (*array-like, integer*) – Pair-counts for the cross-correlation between D1 and D2
- **D1R2** (*array-like, integer*) – Pair-counts for the cross-correlation between D1 and R2

- **D2R1** (*array-like, integer*) – Pair-counts for the cross-correlation between D2 and R1
- **R1R2** (*array-like, integer*) – Pair-counts for the cross-correlation between R1 and R2
- **all of these pair-counts arrays, the corresponding numpy** (*For*) –
- **returned by the theory/mocks modules can also be passed** (*struct*) –
- **estimator** (*string, default='LS' (Landy-Szalay)*) – The kind of estimator to use for computing the correlation function. Currently, only supports Landy-Szalay

Returns **cf** – The correlation function, calculated using the chosen estimator, is returned. NAN is returned for the bins where the RR count is 0.

Return type A numpy array

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from Corrfunc.theory.DD import DD
>>> from Corrfunc.io import read_catalog
>>> from Corrfunc.utils import convert_3d_counts_to_cf
>>> X, Y, Z = read_catalog()
>>> N = len(X)
>>> boxsize = 420.0
>>> rand_N = 3*N
>>> seed = 42
>>> np.random.seed(seed)
>>> rand_X = np.random.uniform(0, boxsize, rand_N)
>>> rand_Y = np.random.uniform(0, boxsize, rand_N)
>>> rand_Z = np.random.uniform(0, boxsize, rand_N)
>>> nthreads = 2
>>> rmin = 0.1
>>> rmax = 15.0
>>> nbins = 10
>>> bins = np.linspace(rmin, rmax, nbins + 1)
>>> autocorr = 1
>>> DD_counts = DD(autocorr, nthreads, bins, X, Y, Z)
>>> autocorr = 0
>>> DR_counts = DD(autocorr, nthreads, bins,
...                X, Y, Z,
...                X2=rand_X, Y2=rand_Y, Z2=rand_Z)
>>> autocorr = 1
>>> RR_counts = DD(autocorr, nthreads, bins, rand_X, rand_Y, rand_Z)
>>> cf = convert_3d_counts_to_cf(N, N, rand_N, rand_N,
...                              DD_counts, DR_counts,
...                              DR_counts, RR_counts)
>>> for xi in cf: print("{0:10.6f}".format(xi))
...
22.769019
 3.612709
 1.621372
 1.000969
```

(continues on next page)

(continued from previous page)

```
0.691646
0.511819
0.398872
0.318815
0.255643
0.207759
```

Corrfunc.utils.**convert_rp_pi_counts_to_wp**(*ND1*, *ND2*, *NR1*, *NR2*, *D1D2*, *D1R2*, *D2R1*, *R1R2*, *nrpbins*, *pimax*, *dpi=1.0*, *estimator=u'LS'*)

Converts raw pair counts to a correlation function.

Parameters

- **ND1** (*integer*) – Number of points in the first dataset
- **ND2** (*integer*) – Number of points in the second dataset
- **NR1** (*integer*) – Number of points in the randoms for first dataset
- **NR2** (*integer*) – Number of points in the randoms for second dataset
- **D1D2** (*array-like, integer*) – Pair-counts for the cross-correlation between D1 and D2
- **D1R2** (*array-like, integer*) – Pair-counts for the cross-correlation between D1 and R2
- **D2R1** (*array-like, integer*) – Pair-counts for the cross-correlation between D2 and R1
- **R1R2** (*array-like, integer*) – Pair-counts for the cross-correlation between R1 and R2
- **all of these pair-counts arrays, the corresponding numpy** (*For*) –
- **returned by the theory/mocks modules can also be passed** (*struct*) –
- **nrpbins** (*integer*) – Number of bins in `rp`
- **pimax** (*float*) – Integration distance along the line of sight direction
- **dpi** (*float, default=1.0 Mpc/h*) – Binsize in the line of sight direction
- **estimator** (*string, default='LS' (Landy-Szalay)*) – The kind of estimator to use for computing the correlation function. Currently, only supports Landy-Szalay

Returns **wp** – The projected correlation function, calculated using the chosen estimator, is returned. If *any* of the `pi` bins (in an `rp` bin) contains 0 for the `RR` counts, then `NAN` is returned for that `rp` bin.

Return type A numpy array

Example

```
>>> from __future__ import print_function
>>> import numpy as np
>>> from Corrfunc.theory.DDrppi import DDrppi
```

(continues on next page)

(continued from previous page)

```
>>> from Corrfunc.io import read_catalog
>>> from Corrfunc.utils import convert_rp_pi_counts_to_wp
>>> X, Y, Z = read_catalog()
>>> N = len(X)
>>> boxsize = 420.0
>>> rand_N = 3*N
>>> seed = 42
>>> np.random.seed(seed)
>>> rand_X = np.random.uniform(0, boxsize, rand_N)
>>> rand_Y = np.random.uniform(0, boxsize, rand_N)
>>> rand_Z = np.random.uniform(0, boxsize, rand_N)
>>> nthreads = 4
>>> pimax = 40.0
>>> nrpbins = 20
>>> rpmin = 0.1
>>> rpmax = 10.0
>>> bins = np.linspace(rpmin, rpmax, nrpbins + 1)
>>> autocorr = 1
>>> DD_counts = DDrppi(autocorr, nthreads, pimax, bins,
...                    X, Y, Z)
>>> autocorr = 0
>>> DR_counts = DDrppi(autocorr, nthreads, pimax, bins,
...                    X, Y, Z,
...                    X2=rand_X, Y2=rand_Y, Z2=rand_Z)
>>> autocorr = 1
>>> RR_counts = DDrppi(autocorr, nthreads, pimax, bins,
...                    rand_X, rand_Y, rand_Z)
>>> wp = convert_rp_pi_counts_to_wp(N, N, rand_N, rand_N,
...                                 DD_counts, DR_counts,
...                                 DR_counts, RR_counts,
...                                 nrpbins, pimax)
>>> for w in wp: print("{0:10.6f}".format(w))
...
187.592199
 83.059181
 53.200599
 40.389354
 33.356371
 29.045476
 26.088133
 23.628340
 21.703961
 20.153125
 18.724781
 17.433235
 16.287183
 15.443230
 14.436193
 13.592727
 12.921226
 12.330074
 11.696364
 11.208365
```

Corrfunc.utils.**translate_isa_string_to_enum**(*isa*)

Helper function to convert an user-supplied string to the underlying enum in the C-API. The extensions only have specific implementations for AVX, SSE42 and FALLBACK. Any other value will raise a ValueError.

Parameters **isa** (*string*) – A string containing the desired instruction set. Valid values are ['AVX', 'SSE42', 'FALLBACK', 'FASTEST']

Returns **instruction_set** – An integer corresponding to the desired instruction set, as used in the underlying C API. The enum used here should be defined *exactly* the same way as the enum in `utils/defs.h`.

Return type integer

`Corrfunc.utils.`**`return_file_with_rbins`**(*rbins*)

Helper function to ensure that the `binfile` required by the Corrfunc extensions is a actually a string.

Checks if the input is a string and file; return if True. If not, and the input is an array, then a temporary file is created and the contents of rbins is written out.

Parameters **rbins** (*string or array-like*) – Expected to be a string or an array containing the bins

Returns **binfile** – If the input `rbins` was a valid filename, then returns the same string. If `rbins` was an array, then this function creates a temporary file with the contents of the `rbins` arrays. This temporary filename is returned

Return type string, filename

`Corrfunc.utils.`**`fix_ra_dec`**(*ra*, *dec*)

Wraps input RA and DEC values into range expected by the extensions.

Parameters

- **RA** (*array-like, units must be degrees*) – Right Ascension values (astronomical longitude)
- **DEC** (*array-like, units must be degrees*) – Declination values (astronomical latitude)

Returns **Tuple (RA, DEC)** – RA is wrapped into range [0.0, 360.0] Declination is wrapped into range [-90.0, 90.0]

Return type array-like

`Corrfunc.utils.`**`fix_cz`**(*cz*)

Multiplies the input array by speed of light, if the input values are too small.

Essentially, converts redshift into *cz*, if the user passed redshifts instead of *cz*.

Parameters **cz** (*array-like, reals*) – An array containing `[Speed of Light *] redshift` values.

Returns **cz** – Actual `cz` values, multiplying the input `cz` array by the `Speed of Light`, if `redshift` values were passed as input `cz`.

Return type array-like

`Corrfunc.utils.`**`compute_nbins`**(*max_diff*, *binsize*, *refine_factor=1*, *max_nbins=None*)

Helper utility to find the number of bins for that satisfies the constraints of (binsize, refine_factor, and max_nbins).

Parameters

- **max_diff** (*double*) – Max. difference (spatial or angular) to be spanned, (i.e., range of allowed domain values)
- **binsize** (*double*) – Min. allowed binsize (spatial or angular)

- **refine_factor** (*integer, default 1*) – How many times to refine the bins. The refinements occurs after `nbins` has already been determined (with `refine_factor-1`). Thus, the number of bins will be **exactly** higher by `refine_factor` compared to the base case of `refine_factor=1`
- **max_nbins** (*integer, default None*) – Max number of allowed cells

Returns **nbins** – Number of bins that satisfies the constraints of bin size >= `binsize`, the refinement factor and nbins <= `max_nbins`.

Return type integer, >= 1

Example

```
>>> from Corrfunc.utils import compute_nbins
>>> max_diff = 180
>>> binsize = 10
>>> compute_nbins(max_diff, binsize)
18
>>> refine_factor=2
>>> max_nbins = 20
>>> compute_nbins(max_diff, binsize, refine_factor=refine_factor,
...               max_nbins=max_nbins)
20
```

Corrfunc.utils.**gridlink_sphere**(*thetamax*, *ra_limits=None*, *dec_limits=None*, *link_in_ra=True*, *ra_refine_factor=1*, *dec_refine_factor=1*, *max_ra_cells=100*, *max_dec_cells=200*, *return_num_ra_cells=False*, *input_in_degrees=True*)

A method to optimally partition spherical regions such that pairs of points within a certain angular separation, `thetamax`, can be quickly computed.

Generates the binning scheme used in *Corrfunc.mocks.DDtheta_mocks* for a spherical region in Right Ascension (RA), Declination (DEC) and a maximum angular separation.

For a given `thetamax`, regions on the sphere are divided into bands in DEC bands, with the width in DEC equal to `thetamax`. If `link_in_ra` is set, then these DEC bands are further sub-divided into RA cells.

Parameters

- **thetamax** (*double*) – Max. angular separation of pairs. Expected to be in degrees unless `input_in_degrees` is set to `False`.
- **ra_limits** (*array of 2 doubles. Default [0.0, 2*pi]*) – Range of Righ Ascension (longitude) for the spherical region
- **dec_limits** (*array of 2 doubles. Default [-pi/2, pi/2]*) – Range of Declination (latitude) values for the spherical region
- **link_in_ra** (*Boolean. Default True*) – Whether linking in RA is done (in addition to linking in DEC)
- **ra_refine_factor** (*integer, >= 1. Default 1*) – Controls the sub-division of the RA cells. For a large number of particles, higher *ra_refine_factor* typically results in a faster runtime
- **dec_refine_factor** (*integer, >= 1. Default 1*) – Controls the sub-division of the DEC cells. For a large number of particles, higher *dec_refine_factor* typically results in a faster runtime

- **max_ra_cells** (*integer, >= 1. Default 100*) – The max. number of RA cells **per DEC band**.
- **max_dec_cells** (*integer >= 1. Default 200*) – The max. number of total DEC bands
- **return_num_ra_cells** (*bool, default False*) – Flag to return the number of RA cells per DEC band
- **input_in_degrees** (*Boolean. Default True*) – Flag to show if the input quantities are in degrees. If set to False, all angle inputs will be taken to be in radians.

Returns

- **sphere_grid** (*A numpy compound array, shape (ncells, 2)*) – A numpy compound array with fields `dec_limit` and `ra_limit` of size 2 each. These arrays contain the beginning and end of DEC and RA regions for the cell.
- **num_ra_cells** (numpy array, returned if `return_num_ra_cells` is set) – A numpy array containing the number of RA cells per declination band

---

Note: If `link_in_ra=False`, then there is effectively one RA bin per DEC band. The 'ra_limit' field will show the range of allowed RA values.

See also:

Corrfunc.mocks.DDtheta_mocks

Example

```
>>> from Corrfunc.utils import gridlink_sphere
>>> import numpy as np
>>> np.set_printoptions(precision=8)
>>> thetamax=30
>>> grid = gridlink_sphere(thetamax)
>>> print(grid)
[([-1.57079633, -1.04719755], [ 0.        ,  3.14159265])
 ([-1.57079633, -1.04719755], [ 3.14159265,  6.28318531])
 ([-1.04719755, -0.52359878], [ 0.        ,  3.14159265])
 ([-1.04719755, -0.52359878], [ 3.14159265,  6.28318531])
 ([-0.52359878,  0.        ], [ 0.        ,  1.25663706])
 ([-0.52359878,  0.        ], [ 1.25663706,  2.51327412])
 ([-0.52359878,  0.        ], [ 2.51327412,  3.76991118])
 ([-0.52359878,  0.        ], [ 3.76991118,  5.02654825])
 ([-0.52359878,  0.        ], [ 5.02654825,  6.28318531])
 ([ 0.        ,  0.52359878], [ 0.        ,  1.25663706])
 ([ 0.        ,  0.52359878], [ 1.25663706,  2.51327412])
 ([ 0.        ,  0.52359878], [ 2.51327412,  3.76991118])
 ([ 0.        ,  0.52359878], [ 3.76991118,  5.02654825])
 ([ 0.        ,  0.52359878], [ 5.02654825,  6.28318531])
 ([ 0.52359878,  1.04719755], [ 0.        ,  3.14159265])
 ([ 0.52359878,  1.04719755], [ 3.14159265,  6.28318531])
 ([ 1.04719755,  1.57079633], [ 0.        ,  3.14159265])
 ([ 1.04719755,  1.57079633], [ 3.14159265,  6.28318531])]
>>> grid = gridlink_sphere(60, dec_refine_factor=3, ra_refine_factor=2)
>>> print(grid)
[([-1.57079633, -1.22173048], [ 0.        ,  1.57079633])
```

(continues on next page)

(continued from previous page)

```
([-1.57079633, -1.22173048], [ 1.57079633,  3.14159265])
([-1.57079633, -1.22173048], [ 3.14159265,  4.71238898])
([-1.57079633, -1.22173048], [ 4.71238898,  6.28318531])
([-1.22173048, -0.87266463], [ 0.        ,  1.57079633])
([-1.22173048, -0.87266463], [ 1.57079633,  3.14159265])
([-1.22173048, -0.87266463], [ 3.14159265,  4.71238898])
([-1.22173048, -0.87266463], [ 4.71238898,  6.28318531])
([-0.87266463, -0.52359878], [ 0.        ,  1.57079633])
([-0.87266463, -0.52359878], [ 1.57079633,  3.14159265])
([-0.87266463, -0.52359878], [ 3.14159265,  4.71238898])
([-0.87266463, -0.52359878], [ 4.71238898,  6.28318531])
([-0.52359878, -0.17453293], [ 0.        ,  1.57079633])
([-0.52359878, -0.17453293], [ 1.57079633,  3.14159265])
([-0.52359878, -0.17453293], [ 3.14159265,  4.71238898])
([-0.52359878, -0.17453293], [ 4.71238898,  6.28318531])
([-0.17453293,  0.17453293], [ 0.        ,  1.57079633])
([-0.17453293,  0.17453293], [ 1.57079633,  3.14159265])
([-0.17453293,  0.17453293], [ 3.14159265,  4.71238898])
([-0.17453293,  0.17453293], [ 4.71238898,  6.28318531])
([ 0.17453293,  0.52359878], [ 0.        ,  1.57079633])
([ 0.17453293,  0.52359878], [ 1.57079633,  3.14159265])
([ 0.17453293,  0.52359878], [ 3.14159265,  4.71238898])
([ 0.17453293,  0.52359878], [ 4.71238898,  6.28318531])
([ 0.52359878,  0.87266463], [ 0.        ,  1.57079633])
([ 0.52359878,  0.87266463], [ 1.57079633,  3.14159265])
([ 0.52359878,  0.87266463], [ 3.14159265,  4.71238898])
([ 0.52359878,  0.87266463], [ 4.71238898,  6.28318531])
([ 0.87266463,  1.22173048], [ 0.        ,  1.57079633])
([ 0.87266463,  1.22173048], [ 1.57079633,  3.14159265])
([ 0.87266463,  1.22173048], [ 3.14159265,  4.71238898])
([ 0.87266463,  1.22173048], [ 4.71238898,  6.28318531])
([ 1.22173048,  1.57079633], [ 0.        ,  1.57079633])
([ 1.22173048,  1.57079633], [ 1.57079633,  3.14159265])
([ 1.22173048,  1.57079633], [ 3.14159265,  4.71238898])
([ 1.22173048,  1.57079633], [ 4.71238898,  6.28318531])]
```

CHAPTER 3

License and Credits

Python Module Index

C

Index

X