nbodykit Documentation

Release 0.1.11

Yu Feng, Nick Hand

June 11, 2017

1 Documentation **3**

2 Get in touch **159**

Python Module Index **161**

nbodykit is an open source project and Python package providing a set of algorithms useful in the analysis of cosmological datasets from N-body simulations and large-scale structure surveys.

Driven by the optimism regarding the abundance and availability of large-scale computing resources in the future, the development of nbodykit distinguishes itself from other similar software packages (i.e., nbodyshop, pynbody, yt, xi) by focusing on :

- a **unified** treatment of simulation and observational datasets by insulating algorithms from data containers
- reducing wall-clock time by **scaling** to thousands of cores
- **deployment** and availability on large, super computing facilities

All algorithms are parallel and run with Message Passing Interface (MPI).

For users using the NERSC super-computers, we provide a ready-to-use tarball of nbodykit and its dependencies; see *Using nbodykit on NERSC* for more details.

CHAPTER 1

Documentation

Installation

Required dependencies

The well-established dependencies are:

- Python 2.7 or 3.4
- scipy, numpy : the foundations for scientific Python
- mpi4py : MPI for Python
- h5py : support for HDF5 files in Python

with a suite of additional tools:

- astropy : a community Python library for astronomy
- pfft-python : a Python binding of pfft, a massively parallel Fast Fourier Transform implementation with pencil domains
- pmesh : a particle mesh framework in Python
- kdcount : pair-counting and Friends-of-Friends clustering with KD-Tree
- bigfile : a reproducible, massively parallel IO library for hierarchical data
- MP-sort : massively parallel sorting
- sharedmem : in-node parallelism with fork and copy-on-write

Optional dependencies

For reading data using pandas

- pandas and pytables are required to use the `Pandas` DataSource, which uses *pandas* for fast parsing of plain text files, as well as the *pandas* subset of HDF5

For creating data using a Halo Occupation Distribution

- halotools is required to use the `Zheng07HOD` DataSource, which provides a general framework for populating halos with galaxies using Halo Occupation Distribution modeling

For generating simulated data from linear power spectra

- classylss, a python binding of the Boltzmann code CLASS, is required to use the `ZeldovichSim` DataSource, which computes a simulated data catalog using the Zel'dovich approximation (from a linear power spectrum computed with CLASS)

Instructions

The software is designed to be installed with the `pip` utility like a regular Python package. The first step on all supported platforms is to checkout the source code via:

```
git clone http://github.com/bccp/nbodykit
cd nbodykit
```

Linux

The steps listed below are intended for a commodity Linux-based cluster (e.g., a Rocks cluster) or a Linux-based workstation / laptop.

To install the main nbodykit package, as well as the external dependencies listed above, into the default Python installation directory:

```
pip install -r requirements.txt
pip install -U --force --no-deps .
```

A different installation directory can be specified via the `--user` or `--root <dir>` options of the `pip install` command.

Mac OS X

The `autotools` software is needed on Mac:

```
sudo port install autoconf automake libtool
```

Using recent versions of MacPorts, we also need to tell `mpicc` to use `gcc` rather than the default `clang` compiler, which doesn't compile `fftw` correctly due to the lack of `openmp` support. Additionally, the `LDSHARED` environment variable must be explicitly set.

In bash, the installation command is:

```
export OMPI_CC=gcc
export LDSHARED="mpicc -bundle -undefined dynamic_lookup -DOMPI_IMPORTS"; pip install -r requirements
pip install -U --force --no-deps .
```

Development Mode

nbodykit can be installed with the development mode (`-e`) of pip

```
pip install -r requirements.txt -e .
```

In addition to the dependency packages, the 'development' installation of nbodykit may require a forced update from time to time:

```
pip install -U --force --no-deps -e .
```

It is sometimes required to manually remove the `nbodykit` directory in `site-packages`, if the above command does not appear to update the installation as expected.

Final Notes

The dependencies of nbodykit are not fully stable, thus we recommend updating the external dependencies occassionally via the `-U` option of `pip install`. Also, the `--force` option ensures that the current sourced version is installed:

```
pip install -U -r requirements.txt
pip install -U --force --no-deps .
```

To confirm that nbodykit is working, we can type, in a interactive Python session:

```
import nbodykit
print(nbodykit)

import kdcount
print(kdcount)

import pmesh
print(pmesh)
```

Or try the scripts in the bin directory:

```
cd bin/
mpirun -n 4 python nbkit.py -h
```

To run the test suite after installing nbodykit, install py.test and pytest-pipeline and run `py.test nbodykit` from the base directory of the source code:

```
pip install pytest pytest-pipeline
pytest nbodykit
```

Using nbodykit on NERSC

In this section, we give instructions for using the latest stable build of nbodykit on NERSC machines (Edison and Cori), which is provided ready-to-use and is recommended for first-time users. For more advanced users, we also provide instructions for performing active development of the source code on NERSC.

When using nbodykit on NERSC, we need to ensure that the Python environment is set up to work efficiently on the computing nodes. The default Python start-up time scales badly with the number of processes, so we employ the python-mpi-bcast tool to ensure fast and reliable start-up times when using nbodykit. This tool can be accessed on both the Cori and Edison machines.

General Usage

We maintain a daily build of the latest stable version of nbodykit on NERSC systems that works with the `2.7-anaconda` Python module and uses the *python-mpi-bcast* helper tool for fast startup of Python. Please see this tutorial for further details about using *python-mpi-bcast* to launch Python applications on NERSC.

In addition to up-to-date builds of nbodykit, we provide a tool (`/usr/common/contrib/bccp/nbodykit/activate.sh`) designed to be used in job scripts to automatically load nbodykit and ensure a fast startup time using *python-mpi-bcast*.

Below is an example job script that prints the help message of the `FFTPower` algorithm:

```
#!/bin/bash
#SBATCH -p debug
#SBATCH -o nbkit-example
#SBATCH -n 16

# You can also allocate the nodes with salloc
#
# salloc -n 16
#
# and type the commands in the shell obtained from salloc

module unload python
module load python/2.7-anaconda

source /usr/common/contrib/bccp/nbodykit/activate.sh

# regular nbodykit command lines
# replace nbkit.py with srun-nbkit

srun-nbkit -n 16 FFTPower --help

# You can also do this in an interactive shell
# e.g.
```

Active development

If you would like to use your own development version of nbodykit directly on NERSC, more installation work is required, although we also provide tools to simplify this process.

We can divide the addititional work into 3 separate steps:

1. When building nbodykit on a NERSC machine, we need to ensure the Python environment is set up to work efficiently on the computing nodes.

If darshan or altd are loaded by default, be sure to unload them before installing, as they tend to interfere with Python:

```
module unload darshan
module unload altd
```

and preferentially, use GNU compilers from PrgEnv-gnu

```
module unload PrgEnv-intel
module unload PrgEnv-cray
module load PrgEnv-gnu
```

then load the Anaconda Python distribution,

```
module load python/2.7-anaconda
```

For convenience, these lines can be included in the shell profile configuration file on NERSC (i.e., `~/.bash_profile.ext`).

2. For easy loading of nbodykit on the compute nodes, we provide tools to create separate bundles (tarballs) of the nbodykit source code and dependencies. This can be performed using the `build.sh` script in the `nbodykit/nersc` directory in the source code tree.

```
cd nbodykit/nersc;

# build the dependencies into a bundle
# this creates the file `$NERSC_HOST/nbodykit-dep.tar.gz`
bash build.sh deps

# build the source code into a separate bundle
# this creates the file `$NERSC_HOST/nbodykit.tar.gz`
bash build.sh source
```

When the source code changes or the dependencies need to be updated, simply repeat the relevant `build.sh` command given above to regenerate the bundle.

3. Finally, in the job script, we must explicitly activate *python-mpi-bcast* and load the nbodykit bundles.

```
#!/bin/bash
#SBATCH -p debug
#SBATCH -o nbkit-dev-example
#SBATCH -n 16

# load anaconda
module unload python
module load python/2.7-anaconda

# activate python-mpi-bcast
source /usr/common/contrib/bccp/python-mpi-bcast/nersc/activate.sh

# go to the nbodykit source directory
cd /path/to/nbodykit

# bcast the nbodykit tarballs
bcast nersc/$NERSC_HOST/nbodykit-dep.tar.gz nersc/$NERSC_HOST/nbodykit.tar.gz

# run the main nbodykit executable
srun -n 16 python-mpi /dev/shm/local/bin/nbkit.py FFTPower --help
```

Overview

nbodykit aims to take advantage of the wealth of large-scale computing resources by providing a massively-parallel toolkit to tackle a wide range of problems that arise in the analysis of large-scale structure datasets.

A major goal of the project is to provide a unified treatment of both simulation and observational datasets, allowing nbodykit to be used in the analysis of not only N-body simulations, but also data from current and future large-scale structure surveys.

nbodykit implements a framework that insulates analysis algorithms from data containers by relying on **plugins** that interact with the core of the code base through distinct **extension points**. Such a framework allows the user to create plugins designed for a specific task, which can then be easily loaded by nbodykit, provided that the plugin implements the minimal interface required by the desired extension point.

We provide several built-in extension points and plugins, which we outline below. For more detailed instructions on how to add new plugins to nbodykit, see *Extending nbodykit*.

Extension Points

There are several built-in extension points, which can be found in the *nbodykit.core* module. These classes serve as the mount point for plugins, connecting the core of the nbodykit package to the individual plugin classes. Each extension point defines a specific interface that all plugins of that type must implement.

There are four built-in extension points. Each extension point carries a *registry*, which stores all plugins of that type that have been successfully loaded by the main nbodykit code.

1. **Algorithm**
   - **location**: *nbodykit.core.Algorithm*
   - **registry**: `nbodykit.algorithms`
   - **description**: the mount point for plugins that run one of the high-level algorithms, i.e, a power spectrum calculation or Friends-of-Friends halo finder
2. **DataSource**
   - **location**: *nbodykit.core.DataSource*
   - **registry**: `nbodykit.datasources`
   - **description**: the mount point for plugins that refer to the reading of input data files
3. **Painter**
   - **location**: *nbodykit.core.Painter*
   - **registry**: `nbodykit.painters`
   - **description**: the mount point for plugins that "paint" input data files, where painting refers to the process of gridding a desired quantity on a mesh; the most common example is gridding the density field of a catalog of objects
4. **Transfer**
   - **location**: *nbodykit.core.Transfer*
   - **registry**: `nbodykit.transfers`
   - **description**: the mount point for plugins that apply a kernel to the painted field in Fourier space during power spectrum calculations

Plugins

Plugins are subclasses of an extension point that are designed to handle a specific task, such as reading a certain type of data file, or computing a specific type of algorithm.

The core of the nbodykit functionality comes from the built-in plugins, of which there are numerous. Below, we list each of the built-in plugins and a brief desciption of the class. For further details, the name of each plugin provides a link to the API reference for each class.

1. **Algorithms**

- *BianchiFFTPower*: power spectrum multipoles using FFTs for a data survey with non-trivial geometry, as detailed in Bianchi et al. 2015 (1505.05341)
- *Describe*: describe a specific column of the input DataSource
- *FFTCorrelation*: correlation spectrum calculator via FFT in a periodic box
- *FFTPower*: periodic power spectrum calculator via FFT

- `FOF`: a Friends-of-Friends (FOF) halo finder
- `FOF6D`: finding subhalos from FOF groups; a variant of FOF6D
- `FiberCollisions`: the application of fiber collisions to a galaxy survey
- `PaintGrid`: periodic power spectrum calculator via FFT
- `PairCountCorrelation`: correlation function calculator via pair counting
- `Play`: describe a specific column of the input DataSource
- `RedshiftHistogram`: compute n(z) from the input DataSource
- `Subsample`: create a subsample from a DataSource, and evaluate density (1 + delta) smoothed at the given scale
- `TestBoxSize`: test if all objects in a DataSource fit within a specified BoxSize
- `TidalTensor`: compute the tidal force tensor
- `TraceHalo`: calculate the halo property based on a different set of halo labels.

2. **DataSource**

- `FOFGroups`: read data from a HDF5 FOFGroup file
- `FastPM`: read snapshot files of the FastPM simulation
- `Gadget`: read a flavor of Gadget 2 files (experimental)
- `GadgetGroupTab`: read a flavor of Gadget 2 FOF catalogs (experimental)
- `HaloLabel`: read a file of halo labels (halo id per particle), as generated the FOF algorithm
- `MultiFile`: read snapshot files a multitype file
- `Pandas`: read data from a plaintext or HDF5 file using Pandas
- `PlainText`: read data from a plaintext file using numpy
- `RaDecRedshift`: read (ra, dec, z) from a plaintext file, returning Cartesian coordinates
- `ShiftedObserver`: establish an explicit observer (outside the box) for a periodic box
- `Subsample`: read data from a HDF5 Subsample file
- `TPMLabel`: read file of halo labels as generated from Martin White's TPM
- `TPMSnapshot`: read snapshot files from Martin White's TPM
- `UniformBox`: data particles with uniform positions and velocities
- `ZeldovichSim`: simulated particles using the Zel'dovich approximation
- `Zheng07Hod`: populate an input halo catalog with galaxies using the Zheng et al. 2007 HOD

3. **Painter**

- `DefaultPainter`: grid the density field of an input DataSource of objects, optionally using a weight for each object.
- `MomentumPainter`: grid the velocity-weighted density field (momentum) field of an input DataSource of objects

4. **Transfer**

- `AnisotropicCIC`: divide by a Fourier-space kernel to account for the CIC gridding window function; see Jing et al 2005 (arxiv:0409240)

- *AnisotropicTSC*: divide by a Fourier-space kernel to account for the TSC gridding window function; see Jing et al 2005 (arxiv:0409240)
- *CICWindow*: divide by a Fourier-space kernel to account for the CIC gridding window function; see Jing et al 2005 (arxiv:0409240)
- *NormalizeDC*: normalize the DC amplitude in Fourier space, which effectively divides by the mean in configuration space
- *RemoveDC*: remove the DC amplitude in Fourier space, which sets the mean of the field in configuration space to zero
- *TSCWindow*: divide by a Fourier-space kernel to account for the TSC gridding window function; see Jing et al 2005 (arxiv:0409240)

Running an Algorithm

An nbodykit *Algorithm* can be run using the nbkit.py executable in the `bin` directory. The user can ask for help with the calling signature of the script in the usual way:

```
python bin/nbkit.py -h
```

The intended usage is:

```
python bin/nbkit.py AlgorithmName ConfigFilename
```

The first argument gives the name of the algorithm plugin that the user wishes to execute, while the second argument gives the name of the file to read configuration parameters from (if no file name is given, the script will read from standard input). For a discussion of the parsing of configuration files, see *Writing configuration files*.

Note: If no configuration file is supplied to `nbkit.py`, the code will attempt to read the configuration file from standard input. See *Reading configuration from stdin* for further details.

The `nbkit.py` script also provides an interface for getting help on extension points and individual plugins. A list of the configuration parameters for the built-in plugins of each extension point can be accessed by:

```
# prints help for all DataSource plugins
python bin/nbkit.py --list-datasources

# prints help for all Algorithm plugins
python bin/nbkit.py --list-algorithms

# prints help for all Painter plugins
python bin/nbkit.py --list-painters

# prints help for all Transfer plugins
python bin/nbkit.py --list-transfers
```

and the help message for an individual plugin can be printed by passing the plugin name to the `--list-*` option, i.e.,

```
# prints help message for only the FFTPower algorithm
python bin/nbkit.py --list-algorithms FFTPower
```

will print the help message for the *FFTPower* algorithm. Similarly, the help messages for specific algorithms can also be accessed by passing the algorithm name and the `-h` option:

```
python bin/nbkit.py FFTPower -h
```

Using MPI

The nbodykit is designed to be run in parallel using the Message Passage Interface (MPI) and the python package mpi4py. The executable `nbkit.py` can take advantage of multiple processors to run algorithms in parallel. The usage for running with *n* processors is:

```
mpirun -n [n] python bin/nbkit.py ...
```

Writing configuration files

The parameters needed to execute the desired algorithms should be stored in a file and passed to the `nbkit.py` file as the second argument. The configuration file should be written using YAML, which relies on the `name:  value` syntax to parse (key, value) pairs into dictionaries in Python.

By example

The YAML syntax is best learned by example. Let's consider the *`FFTPower`* algorithm, which computes the power spectrum of two data fields using a Fast Fourier Transform in a periodic box. The necessary parameters to initialize and run this algorithm can be accessed from the *schema* attribute of the *FFTPower* class:

```
# import the NameSpace holding the loaded algorithms
In [1]: from nbodykit import algorithms

# can also use algorithms.FFTPower? in IPython
In [2]: print(algorithms.FFTPower.schema)
periodic power spectrum calculator via FFT

Parameters
----------
mode : { '1d', '2d' }
    compute the power as a function of `k` or `k` and `mu`
Nmesh : int
    the number of cells in the gridded mesh
field : FieldType
    first data field; a tuple of (DataSource, Painter, Transfer)
    The 3 subfields are:

        DataSource : DataSource.from_config, GridSource.from_config
            the 1st DataSource; run `nbkit.py --list-datasources` for all options
        Painter : Painter.from_config
            the 1st Painter; run `nbkit.py --list-painters` for all options
        Transfer : Transfer.from_config
            the 1st Transfer chain; run `nbkit.py --list-transfers` for all options

other : FieldType, optional
    the other data field; a tuple of (DataSource, Painter, Transfer)
    The 3 subfields are:

        DataSource : DataSource.from_config, GridSource.from_config
            the 2nd DataSource; run `nbkit.py --list-datasources` for all options
        Painter : Painter.from_config
            the 2nd Painter; run `nbkit.py --list-painters` for all options
        Transfer : Transfer.from_config
            the 2nd Transfer chain; run `nbkit.py --list-transfers` for all options

los : { 'x', 'y', 'z' }, optional
    the line-of-sight direction -- the angle `mu` is defined with respect to (default: z)
Nmu : int, optional
    the number of mu bins to use from mu=[0,1]; if `mode = 1d`, then `Nmu` is set to 1 (default: 5)
dk : float, optional
    the spacing of k bins to use; if not provided, the fundamental mode of the box is used
kmin : float, optional
    the edge of the first `k` bin to use; default is 0 (default: 0.0)
quiet : bool, optional
    silence the logging output (default: False)
poles : int, optional
    if specified, also compute these multipoles from P(k,mu) (default: [])
paintbrush : { 'cic', 'tsc' }, optional
    the density assignment kernel to use when painting; CIC (2nd order) or TSC (3rd order) (default:
comm : optional
    the global MPI communicator
```

An example configuration file for this algorithm is given below. The algorithm reads in two data files using the *FastPM* DataSource and *FOFGroups* DataSource classes and computes the cross power spectrum of the density fields.

```
mode: 1d
Nmesh: 256

cosmo: {Om0: 0.27, H0: 100}

# the first field
field:
    DataSource:
        plugin: FastPM
        path: ${NBKIT_CACHE}/data/fastpm_1.0000
    Painter:
        DefaultPainter
    Transfer:
        [NormalizeDC, RemoveDC, AnisotropicCIC]

# the second field to cross-correlate with
other:
    # datasource
    DataSource:
        FOFGroups:
            path: ${NBKIT_CACHE}/data/fof_ll0.200_1.0000.hdf5
            m0: 10.0

    # painter (can omit this and get same value)
    Painter: DefaultPainter

    # transfers (can omit and get this sequence)
    Transfer: [NormalizeDC, RemoveDC, AnisotropicCIC]

output: ${NBKIT_HOME}/examples/output/test_power_cross.dat
```

The key aspect of YAML syntax for nbodykit configuration files is that parameters listed at a common indent level will be parsed together into a dictionary. This is illustrated explicitly with the *cosmo* keyword in line 4, which could have been equivalently expressed as:

```
cosmo:
    Om0: 0.27
    H0: 100
```

A few other things to note:

- The names of the parameters given in the configuration file must exactly match the names of the attributes listed in the algorithm's *schema*.
- All required parameters must be listed in the configuration file, otherwise the code will raise an exception.
- The *field* and *other* parameters in this example have subfields, named *DataSource*, *Painter*, and *Transfer*. The parameters that are subfields must be indented from the their parent parameters to indicate that they are subfields.
- Environment variables can be used in configuration files, using the syntax `${ENV_VAR}` or `$ENV_VAR`. In the above file, both *NBKIT_CACHE* and *NBKIT_HOME* are assumed to be environment variables.

Plugin representations

A key aspect of the nbodykit code is the use of plugins; representing them properly in configuration files is an important step in becoming a successful nbodykit user.

The function responsible for initializing plugins from their configuration file representation is *`from_config()`*. This function accepts several different ways of representing plugins, and we will illustrate these methods using the previous *configuration file*.

1. The parameters needed to initialize a plugin can be given at a common indent level, and the keyword *plugin* can be used to give the name of the plugin to load. This is illustrated for the *field.DataSource* parameter, which will be loaded into a *`FastPM`* DataSource:

```
field:
    DataSource:
        plugin: FastPM
        path: ${NBKIT_CACHE}/data/fastpm_1.0000
```

2. Rather than using the *plugin* parameter to give the name of the plugin to load, the user can indent the plugin arguments under the name of the plugin, as is illustrated below for the *`FOFGroups`* DataSource:

```
other:
    # datasource
    DataSource:
        FOFGroups:
            path: ${NBKIT_CACHE}/data/fof_ll0.200_1.0000.hdf5
            m0: 10.0
```

3. If the plugin needs no arguments to be intialized, the user can simply use the name of the plugin, as is illustrated below for the *field.Painter* parameter:

```
    Painter:
        DefaultPainter
```

For more examples on how to accurately represent plugins in configuration files, see the myriad of configuration files listed in the `examples` directory of the source code.

Specifying the output file

All configuration files must include the `output` parameter. This parameter gives the name of the output file to which the results of the algorithm will be saved.

The nbkit.py script will raise an exception when the output parameter is not present in the input configuration file.

Specifying the cosmology

For the succesful reading of data using some nbodykit DataSource classes, cosmological parameters must be specified. The desired cosmology should be set in the configuration file, as is done in line 4 of the previous example. A single, global cosmology class will be initialized and passed to all DataSource objects that are created while running the nbodykit code.

The cosmology class is located at *nbodykit.cosmology.Cosmology*, and the syntax for the class is borrowed from astropy.cosmology.wCDM class. The constructor arguments are:

```
In [3]: from nbodykit.cosmology import Cosmology

# can also do ``Cosmology?`` in IPython
In [4]: help(Cosmology.__init__)
Help on function __init__ in module nbodykit.cosmology:

__init__(self, H0=67.6, Om0=0.31, Ob0=0.0486, Ode0=0.69, w0=-1.0, Tcmb0=2.7255, Neff=3.04, m_nu=0.0,
    Initialize self.  See help(type(self)) for accurate signature.
```

Reading configuration from stdin

If no configuration file name is supplied to nbkit.py, the code will attempt to read the configuration from standard input. Note that the syntax for passing information via standard input varies by operating system and shell type, and may not be supported for all operating systems.

An example of such a usage is given in the examples/batch directory and is listed below:

```
DIR=`dirname $0`
cd $DIR
[ -d ../output ] || mkdir ../output

echo testing nbkit.py from STDIN ...
echo Some openmpi implementations are buggy causing this test to hang
echo https://bugzilla.redhat.com/show_bug.cgi?id=1235044
echo use Control-C to stop this one if it hangs.

mpirun -np 2 python ../../bin/nbkit.py FFTPower <<EOF
mode: 1d
Nmesh: 256
output: ${NBKIT_HOME}/examples/output/test_stdin.dat

field:
    DataSource:
       plugin: FastPM
       path: ${NBKIT_CACHE}/data/fastpm_1.0000
    Transfer: [NormalizeDC, RemoveDC, AnisotropicCIC]
EOF
```

Running in batch mode

The nbodykit code also provides a tool to run a specific Algorithm for a set of configuration files, possibly executing the algorithms in parallel. We refer to this as "batch mode" and provide the nbkit-batch.py script in the bin directory for this purpose.

Once again, the `-h` flag will provide the help message for this script; the intended usage is:

```
mpirun -n [n] python bin/nbkit-batch.py [--extras EXTRAS] [--debug] [--use_all_cpus] -i TASKS -c CONF
```

The idea here is that a "template" configuration file can be passed to `nbkit-batch.py` via the `-c` option, and this file should contain special keys that will be formatted using `str.format()` syntax when iterating through a set of configuration files. The names of these keys and the desired values for the keys to take when iterating can be specified by the `-i` option.

Note: The configuration template file in "batch" mode using `nbkit-batch.py` should be passed explicitly with a `-c` option, while for normal usage of `nbkit.py`, the configuration file should be passed as the second positional argument.

By example

Let's consider the following invocation of the `nbkit-batch.py` script:

```
mpirun -np 7 python bin/nbkit-batch.py FFTPower 2 -c examples/batch/test_power_batch.template -i "los
```

In this example, the code is executed using MPI with 7 available processors, and we have set *cpus_per_worker* to 2. The `nbkit-batch.py` script reserves one processor to keep track of the task scheduling (the "master" processor), which means that 6 processors are available for computation. With 2 cpus for each worker, the script is able to use 3 workers to execute *FFTPower* algorithms in parallel. Furthermore, we have asked for 3 task values – the input configuration template will have the *los* key updated with values 'x', 'y', and 'z'. With only three tasks and exactly 3 workers, each task can be computed in parallel simulataneously.

For a closer look at how the task values are updated in the template configuration file, let's examine the template file:

```
cosmo : {Om0: 0.28, H0: 70}

mode: 1d
Nmesh: 256
field:
    DataSource:
        plugin: FastPM
        path: ${NBKIT_CACHE}/data/fastpm_1.0000
        rsd: {los}
    Transfer: [NormalizeDC, RemoveDC, AnisotropicCIC]

los: {los}
output: ${NBKIT_HOME}/examples/output/test_batch_power_fastpm_1d_{los}los_{tag}.dat
```

In this file, we see that there is exactly one task key: *los*. The `{los}` string will be updated with the values given on the command-line ('x', 'y', and 'z'), and the *FFTPower* algorithm will be executed for each of the resulting configuration files. The task keys are formatted using the Python string formatting syntax of `str.format()`.

Lastly, we have also passed a file to the `nbkit-batch.py` script using the `--extras` option. This option allows an arbitrary number of extra string keys to be formatted for each task iteration. In this example, the only "extra" key provided is `{tag}`, and the `extra.template` file looks like:

```
tag = ['task_1', 'task_2', 'task_3']
```

So, when updating *los* to the first task value ('x'), the *tag* key is updated to 'task_1', and the pattern continues for the other tasks. With this configuration, `nbkit-batch.py` will output 3 separate files, named:

- test_batch_power_fastpm_1d_xlos_task_1.dat
- test_batch_power_fastpm_1d_ylos_task_2.dat
- test_batch_power_fastpm_1d_zlos_task_3.dat

Multiple task keys

The `-i` flag can be passed multiple times to the `nbkit-batch.py` script. For example, let us imagine that in addition to the *los* task key, we also wanted to iterate over a *box* key. If we had two boxes, labeled *1* and *2*, then we could also specify `-i box: ['1', '2']` on the command-line. Then, the task values that would be iterated over are:

```
(`los`, `box`) = ('x', '1'), ('x', '2'), ('y', '1'), ('y', '2'), ('z', '1'), ('z', '2')
```

DataSet for Algorithm results

Several nbodykit algorithms compute two-point clustering statistics, and we provide the `DataSet` class for analyzing these results. The class is designed to hold data variables at fixed coordinates, i.e., a grid of (r, μ) or (k, μ) bins.

The DataSet class is modeled after the syntax of `xarray.Dataset`, and there are several subclasses of DataSet that are specifically designed to hold correlation function or power spectrum results (in 1D or 2D).

For algorithms that compute power spectra, we have:

- **`FFTPower`**
  - computes: $P(k, \mu)$ or $P(k)$
  - results class: `nbodykit.dataset.Power2dDataSet` or `nbodykit.dataset.Power1dDataSet`
- **`BianchiFFTPower`**
  - computes: $P(k)$
  - results class: `nbodykit.dataset.Power1dDataSet`

And for algorithms computing correlation functions:

- **`FFTCorrelation`**, **`PairCountCorrelation`**
  - computes: $\xi(k, \mu)$ or $\xi(k)$
  - results class: `nbodykit.dataset.Corr2dDataSet` or `nbodykit.dataset.Corr1dDataSet`

Loading results

To load power spectrum or correlation function results, the user must first read the plaintext files and then initialize the relevant subclass of DataSet. The functions `nbodykit.files.Read2DPlainText()` and `nbodykit.files.Read1DPlainText()` should be used for reading 2D and 1D result files, respectively.

The reading and DataSet initialization can be performed in one step, taking advantage of `from_nbkit()`:

```
In [1]: from nbodykit import dataset, files

# output file of 'examples/power/test_plaintext.params'
In [2]: filename_2d = os.path.join(cache_dir, 'results', 'test_power_plaintext.dat')
```

```
# load a 2D power result
In [3]: power_2d =  dataset.Power2dDataSet.from_nbkit(*files.Read2DPlainText(filename_2d))

In [4]: power_2d
Out[4]: <Power2dDataSet: dims: (k_cen: 128, mu_cen: 5), variables: ('mu', 'k', 'modes', 'power')>

# output file of 'examples/power/test_cross_power.params'
In [5]: filename_1d =  os.path.join(cache_dir, 'results', 'test_power_cross.dat')

# load a 1D power result
In [6]: power_1d =  dataset.Power1dDataSet.from_nbkit(*files.Read1DPlainText(filename_1d))

In [7]: power_1d
Out[7]: <Power1dDataSet: dims: (k_cen: 128), variables: ('k', 'modes', 'power.imag', 'power.real')>
```

Coordinate grid

The clustering statistics are measured for fixed bins, and the DataSet class has several attributes to access the coordinate grid defined by these bins:

- `shape`: the shape of the coordinate grid
- `dims`: the names of each dimension of the coordinate grid
- `coords`: a dictionary that gives the center bin values for each dimension of the grid
- `edges`: a dictionary giving the edges of the bins for each coordinate dimension

```
In [8]: print(power_1d.shape, power_2d.shape)
(128,) (128, 5)

In [9]: print(power_1d.dims, power_2d.dims)
\\\\\\\\\\\\\\\\['k_cen'] ['k_cen', 'mu_cen']

In [10]: power_2d.coords
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[10]:
{'k_cen': array([ 0.00227652,  0.00682955,  0.01138258,  0.01593562,  0.02048864,
          0.02504168,  0.02959472,  0.03414775,  0.03870078,  0.04325382,
          0.04780685,  0.05235988,  0.05691291,  0.06146595,  0.06601898,
          0.07057201,  0.07512505,  0.07967807,  0.08423111,  0.08878414,
          0.09333717,  0.09789019,  0.10244325,  0.1069963 ,  0.1115493 ,
          0.11610235,  0.1206554 ,  0.1252084 ,  0.12976145,  0.1343145 ,
          0.1388675 ,  0.14342055,  0.1479736 ,  0.1525266 ,  0.1570796 ,
          0.16163265,  0.1661857 ,  0.1707387 ,  0.17529175,  0.1798448 ,
          0.1843978 ,  0.18895085,  0.1935039 ,  0.1980569 ,  0.20260995,
          0.207163  ,  0.211716  ,  0.21626905,  0.2208221 ,  0.2253751 ,
          0.22992815,  0.2344812 ,  0.2390342 ,  0.24358725,  0.2481403 ,
          0.2526933 ,  0.25724635,  0.2617994 ,  0.2663524 ,  0.27090545,
          0.2754585 ,  0.2800115 ,  0.28456455,  0.2891176 ,  0.2936706 ,
          0.29822365,  0.3027767 ,  0.3073297 ,  0.31188275,  0.3164358 ,
          0.3209888 ,  0.32554185,  0.3300949 ,  0.3346479 ,  0.33920095,
          0.343754  ,  0.348307  ,  0.35286005,  0.3574131 ,  0.3619661 ,
          0.36651915,  0.3710722 ,  0.3756252 ,  0.38017825,  0.3847313 ,
          0.3892843 ,  0.39383735,  0.3983904 ,  0.4029434 ,  0.40749645,
          0.4120495 ,  0.4166025 ,  0.42115555,  0.4257086 ,  0.4302616 ,
          0.43481465,  0.4393677 ,  0.4439207 ,  0.44847375,  0.4530268 ,
          0.4575798 ,  0.4621328 ,  0.46668585,  0.4712389 ,  0.4757919 ,
          0.48034495,  0.484898  ,  0.489451  ,  0.49400405,  0.4985571 ,
          0.5031101 ,  0.50766315,  0.5122162 ,  0.5167692 ,  0.52132225,
          0.5258753 ,  0.5304283 ,  0.53498135,  0.5395344 ,  0.5440874 ,
          0.54864045,  0.5531935 ,  0.5577465 ,  0.56229955,  0.5668526 ,
          0.5714056 ,  0.57595865,  0.5805117 ]),
  'mu_cen': array([ 0.1,   0.3,   0.5,   0.7,   0.9])}

 In [11]: power_2d.edges
 \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
 {'k_cen': array([ 0.        ,  0.00455303,  0.00910607,  0.0136591 ,  0.01821213,
          0.02276516,  0.0273182 ,  0.03187123,  0.03642426,  0.0409773 ,
          0.04553033,  0.05008336,  0.05463639,  0.05918943,  0.06374246,
          0.06829549,  0.07284853,  0.07740156,  0.08195459,  0.08650762,
          0.09106066,  0.09561369,  0.1001667 ,  0.1047198 ,  0.1092728 ,
          0.1138258 ,  0.1183789 ,  0.1229319 ,  0.1274849 ,  0.132038  ,
          0.136591  ,  0.141144  ,  0.1456971 ,  0.1502501 ,  0.1548031 ,
          0.1593561 ,  0.1639092 ,  0.1684622 ,  0.1730152 ,  0.1775683 ,
          0.1821213 ,  0.1866743 ,  0.1912274 ,  0.1957804 ,  0.2003334 ,
          0.2048865 ,  0.2094395 ,  0.2139925 ,  0.2185456 ,  0.2230986 ,
          0.2276516 ,  0.2322047 ,  0.2367577 ,  0.2413107 ,  0.2458638 ,
          0.2504168 ,  0.2549698 ,  0.2595229 ,  0.2640759 ,  0.2686289 ,
          0.273182  ,  0.277735  ,  0.282288  ,  0.2868411 ,  0.2913941 ,
          0.2959471 ,  0.3005002 ,  0.3050532 ,  0.3096062 ,  0.3141593 ,
          0.3187123 ,  0.3232653 ,  0.3278184 ,  0.3323714 ,  0.3369244 ,
          0.3414775 ,  0.3460305 ,  0.3505835 ,  0.3551366 ,  0.3596896 ,
          0.3642426 ,  0.3687957 ,  0.3733487 ,  0.3779017 ,  0.3824548 ,
          0.3870078 ,  0.3915608 ,  0.3961139 ,  0.4006669 ,  0.4052199 ,
          0.409773  ,  0.414326  ,  0.418879  ,  0.4234321 ,  0.4279851 ,
          0.4325381 ,  0.4370912 ,  0.4416442 ,  0.4461972 ,  0.4507503 ,
          0.4553033 ,  0.4598563 ,  0.4644093 ,  0.4689624 ,  0.4735154 ,
          0.4780684 ,  0.4826215 ,  0.4871745 ,  0.4917275 ,  0.4962806 ,
          0.5008336 ,  0.5053866 ,  0.5099397 ,  0.5144927 ,  0.5190457 ,
          0.5235988 ,  0.5281518 ,  0.5327048 ,  0.5372579 ,  0.5418109 ,
          0.5463639 ,  0.550917  ,  0.55547   ,  0.560023  ,  0.5645761 ,
          0.5691291 ,  0.5736821 ,  0.5782352 ,  0.5827882 ]),
  'mu_cen': array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])}
```

The center bin values can also be directly accessed in a dict-like fashion from the main DataSet using the dimension names:

```
In [12]: power_2d['k_cen'] is power_2d.coords['k_cen']
Out[12]: True

In [13]: power_2d['mu_cen'] is power_2d.coords['mu_cen']
\\\\\\\\\\\\\\Out[13]: True
```

Accessing the data

The names of data variables stored in a DataSet are stored in the `variables` attribute, and the `data` attribute stores the arrays for each of these names in a structured array. The data for a given variable can be accessed in a dict-like fashion:

```
In [14]: power_1d.variables
Out[14]: ['k', 'modes', 'power.imag', 'power.real']

In [15]: power_2d.variables
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[15]: ['mu', 'k', 'modes', 'power']
```

```
# the real component of the power
In [16]: Pk = power_1d['power.real']

In [17]: print(type(Pk), Pk.shape, Pk.dtype)
<class 'numpy.ndarray'> (128,) float64

# complex power array
In [18]: Pkmu = power_2d['power']

In [19]: print(type(Pkmu), Pkmu.shape, Pkmu.dtype)
<class 'numpy.ndarray'> (128, 5) complex128
```

In some cases, the variable value for a given bin will be missing or invalid, which is indicated by a `numpy.nan` value in the `data` array for the given bin. The DataSet class carries a `mask` attribute that defines which elements of the data array have `numpy.nan` values.

Meta-data

An `OrderedDict` of meta-data for a DataSet class is stored in the `attrs` attribute. The *`Read1DPlainText()`* and *`Read2DPlainText()`* functions will load any meta-data saved to file while running an algorithm.

Typically for power spectrum and correlation function results, the `attrs` dictionary stores information about box size, number of objects, etc:

```
In [20]: power_2d.attrs
Out[20]:
OrderedDict([('volume', 2628072000.0),
             ('Lz', 1380.0),
             ('N1', 43956),
             ('Ly', 1380.0),
             ('N2', 43956),
             ('Lx', 1380.0)])
```

To attach additional meta-data to a DataSet class, the user can add additional keywords to the `attrs` dictionary.

Slicing

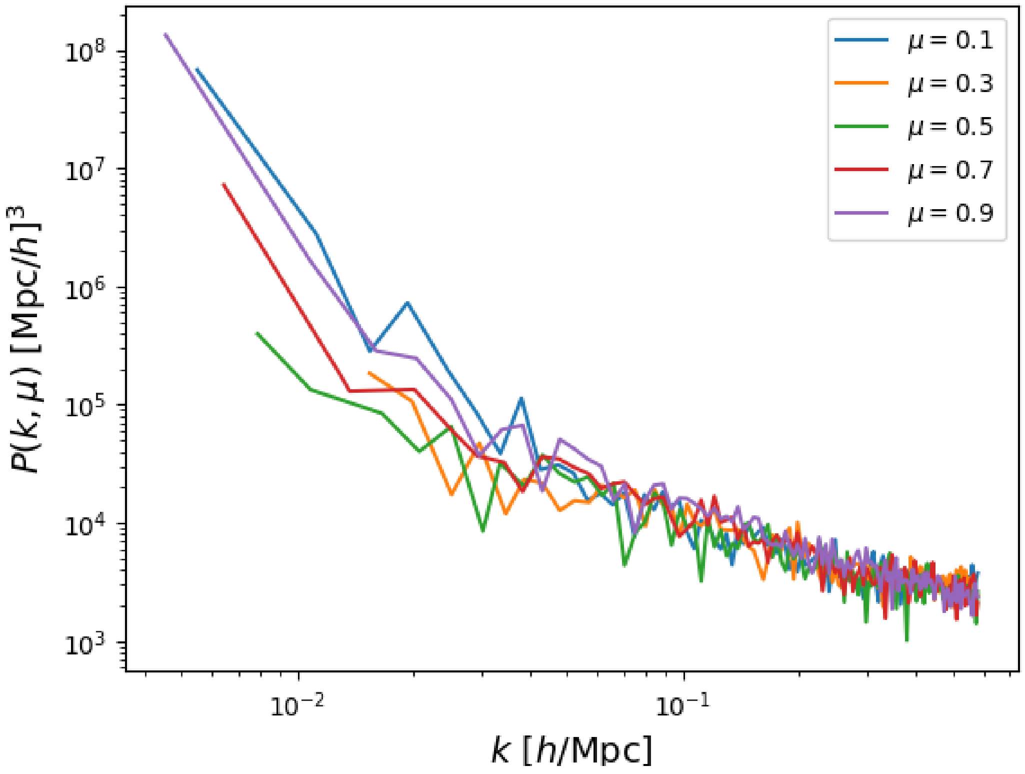
Slices of the coordinate grid of a DataSet can be achieved using array-like indexing of the main DataSet class, which will return a new DataSet holding the sliced data:

```
# select the first mu bin
In [21]: power_2d[:,0]
Out[21]: <Power2dDataSet: dims: (k_cen: 128), variables: ('mu', 'k', 'modes', 'power')>

# select the first and last mu bins
In [22]: power_2d[:, [0, -1]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[22]: <Po

# select the first 5 k bins
In [23]: power_1d[:5]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
```

A typical usage of array-like indexing is to loop over the *mu_cen* dimension of a 2D DataSet, such as when plotting:

```
In [24]: from matplotlib import pyplot as plt
```

```
# the shot noise is volume / number of objects
In [25]: shot_noise = power_2d.attrs['volume'] / power_2d.attrs['N1']

# plot each mu bin separately
In [26]: for i in range(power_2d.shape[1]):
   ....:     pk = power_2d[:,i]
   ....:     label = r"$\mu = %.1f$" % power_2d['mu_cen'][i]
   ....:     plt.loglog(pk['k'], pk['power'].real - shot_noise, label=label)
   ....:

In [27]: print(os.getcwd())
/home/docs/checkouts/readthedocs.org/user_builds/nbodykit/checkouts/0.1.11/docs

In [28]: plt.legend()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[28]: <matplotlib

In [29]: plt.xlabel(r"$k$ [$h$/Mpc]", fontsize=14)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

In [30]: plt.ylabel(r"$P(k,\mu)$ $[\mathrm{Mpc}/h]^3$", fontsize=14)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

In [31]: plt.show()
```



The coordinate grid can also be sliced using label-based indexing, similar to the syntax of `xarray.Dataset.sel()`. The `method` keyword of `sel()` determines if exact coordinate matching is required (`method=None`, the default) or if the nearest grid coordinate should be selected automatically (`method='nearest'`).

For example, we can slice power spectrum results based on *k_cen* and *mu_cen* values:

```
# get all mu bins for the k bin closest to k=0.1
In [32]: power_2d.sel(k_cen=0.1, method='nearest')
Out[32]: <Power2dDataSet: dims: (mu_cen: 5), variables: ('mu', 'k', 'modes', 'power')>

# slice from k=0.01-0.1 for mu = 0.5
In [33]: power_2d.sel(k_cen=slice(0.01, 0.1), mu_cen=0.5, method='nearest')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[33]: <Powe
```

We also provide a function `squeeze()` with functionality similar to `numpy.squeeze()` for the DataSet class:

```
# get all mu bins for the k bin closest to k=0.1, but keep k dimension
In [34]: sliced = power_2d.sel(k_cen=[0.1], method='nearest')

In [35]: sliced
Out[35]: <Power2dDataSet: dims: (k_cen: 1, mu_cen: 5), variables: ('mu', 'k', 'modes', 'power')>

# and then squeeze to remove the k dimension
In [36]: sliced.squeeze()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out
```

Note that, by default, array-based or label-based indexing will automatically "squeeze" sliced objects that have a dimension of length one, unless a list of indexers is used, as is done above.

Reindexing

It is possible to reindex a specific dimension of the coordinate grid using `reindex()`. The new bin spacing must be an integral multiple of the original spacing, and the variable values will be averaged together on the new coordinate grid.

```
In [37]: power_2d.reindex('k_cen', 0.02)
Out[37]: <Power2dDataSet: dims: (k_cen: 32, mu_cen: 5), variables: ('mu', 'k', 'modes', 'power')>

In [38]: power_2d.reindex('mu_cen', 0.4)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out
```

Any variable names passed to `reindex()` via the *fields_to_sum* keyword will have their values summed, instead of averaged, when reindexing. Futhermore, for `Power2dDataSet` and `Power1dDataSet`, the `modes` variable will be automatically summed, and for `Corr2dDataSet` or `Corr1dDataSet`, the `N` and `RR` fields will be automatically summed when reindexing.

Averaging

The average of a specific dimension can be taken using `average()`. A common usage is averaging over the *mu_cen* dimension of a 2D DataSet, which is accomplished by:

```
# compute P(k) from P(k,mu)
In [39]: power_2d.average('mu_cen')
Out[39]: <Power2dDataSet: dims: (k_cen: 128), variables: ('mu', 'k', 'modes', 'power')>
```

Extending nbodykit

One of the goals of the extension point and plugin framework used by nbodykit is to allow the user to easily extend the main code base with new plugins. In this section, we'll describe the details of implementing plugins for the 4 built-in extension points: *Algorithm*, *DataSource*, *Painter*, and *Transfer*.

To define a plugin:

1. Subclass from the desired extension point class.
2. Define a class method *fill_schema* that declares the relevant attributes by calling `add_argument()` of the class's `ConstructorSchema`, which is stored as the *schema* attribute.
3. Define a *plugin_name* class attribute.
4. Define the functions relevant for that extension point interface.

Registering plugins

All plugin classes must define a `fill_schema()` function, which is necessary for the core of the nbodykit code to be aware of and use the plugin class. Each plugin carries a *schema* attribute which is a `nbodykit.plugins.fromfile.ConstructorSchema` that is responsible for storing information regarding the parameters needed to initialize the plugin. The main purpose of `fill_schema()` is to update this schema object for each argument of a plugin's `__init__()`.

As an example of how this is done, we can examine `__init__()` and `fill_schema()` for the *PlainText* DataSource:

```python
    def __init__(self, path, names, BoxSize,
                    usecols=None, poscols=['x','y','z'], velcols=None,
                    rsd=None, posf=1., velf=1., select=None):

        # positional arguments
        self.path = path
        self.names = names
        self.BoxSize = BoxSize

        # keywords
        self.usecols = usecols
        self.poscols = poscols
        self.velcols = velcols
        self.rsd = rsd
        self.posf = posf
        self.velf = velf
        self.select = select
```

```python
    def fill_schema(cls):

        s = cls.schema
        s.description = "read data from a plaintext file using numpy"

        s.add_argument("path", type=str,
            help="the file path to load the data from")
        s.add_argument("names", type=str, nargs='*',
            help="names of columns in text file or name of the data group in hdf5 file")
        s.add_argument("BoxSize", type=cls.BoxSizeParser,
            help="the size of the isotropic box, or the sizes of the 3 box dimensions")
        s.add_argument("usecols", type=str, nargs='*',
            help="only read these columns from file")
        s.add_argument("poscols", type=str, nargs=3,
            help="names of the position columns")
        s.add_argument("velcols", type=str, nargs=3,
            help="names of the velocity columns")
        s.add_argument("rsd", type=str, choices="xyz",
            help="direction to do redshift distortion")
        s.add_argument("posf", type=float,
            help="factor to scale the positions")
        s.add_argument("velf", type=float,
            help="factor to scale the velocities")
        s.add_argument("select", type=selectionlanguage.Query,
            help='row selection based on conditions specified as string, i.e., "Mass > 1e14"')
```

A few things to note in this example:

1. All arguments of `__init__()` are added to the class schema via the `add_argument()` function in the *fill_schema* function.
2. The `add_argument()` function has a calling signature similar to `argparse.ArgumentParser.add_argument()`. The user can specify default values, parameter choices, and type functions used for casting parsed values.
3. Any default values and whether or not the parameter is required will be directly inferred from the `__init__()` calling signature.
4. Parameters in a plugin's schema will be automatically attached to the class instance before the body of `__init__()` is executed – the user does not need to reattach these attributes. As such, the body of `__init__()` in this example is empty. However, additional initialization-related computations could also be performed here.

Extension point interfaces

Below, we provide the help messages for each of the functions that are required to implement plugins for the 4 built-in extension point types.

Algorithm

The functions required to implement an Algorithm plugin are:

```
# run the algorithm
In [1]: help(Algorithm.run)
Help on function run in module nbodykit.core.algorithms:

run(self)
    Run the algorithm

    Returns
    -------
    result : tuple
        the tuple of results that will be passed to :func:`Algorithm.save`


# save the result to an output file
In [2]: help(Algorithm.save)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

save(self, output, result)
    Save the results of the algorithm run

    Parameters
    ----------
    output : str
        the name of the output file to save results too
    result : tuple
        the tuple of results returned by :func:`Algorithm.run`
```

DataSource

The functions required to implement a DataSource plugin are:

```
# read and return all available data columns (recommended for typical users)
In [3]: help(DataSource.readall)
Help on function readall in module nbodykit.core.datasource:

readall(self)
    Override to provide a method to read all available data at once
    (uncollectively) and cache the data in memory for repeated
    calls to `read`

    Notes
    -----
    *   By default, :func:`DataStream.read` calls this function on the
        root rank to read all available data, and then scatters
        the data evenly across all available ranks
    *   The intention is to reduce the complexity of implementing a
        simple and small data source, for which reading all data at once
        is feasible

    Returns
    -------
    data : dict
        a dictionary of all supported data for the data source; keys
        give the column names and values are numpy arrays


# read data columns, reading data in parallel across MPI ranks
In [4]: help(DataSource.parallel_read)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

parallel_read(self, columns, full=False)
    Override this function for complex, large data sets. The read
    operation shall be collective, each yield generates different
    sections of the datasource. No caching of data takes places.

    If the DataSource does not provide a column in `columns`,
    `None` should be returned.

    Notes
    -----
    *   This function will be called if :func:`DataStream.readall` is
        not implemented
    *   The intention is for this function to handle complex and
        large data sets, where parallel I/O across ranks is
        required to avoid memory and I/O issues

    Parameters
    ----------
    columns : list of str
```

```
        the list of data columns to return
    full : bool, optional
        if `True`, any `bunchsize` parameters will be ignored, so
        that each rank will read all of its specified data section
        at once

    Returns
    -------
    data : list
        a list of the data for each column in columns; if the data source
        does not provide a given column, that element should be `None`
```

Painter

The functions required to implement a Painter plugin are:

```
# do the painting procedure
In [5]: help(Painter.paint)
Help on function paint in module nbodykit.core.painter:

paint(self, pm, datasource)
    Paint the DataSource specified to a mesh

    Parameters
    ----------
    pm : :class:`~pmesh.particlemesh.ParticleMesh`
        particle mesh object that does the painting
    datasource : DataSource
        the data source object representing the field to paint onto the mesh

    Returns
    -------
    stats : dict
        dictionary of statistics related to painting and reading of the DataSource
```

Transfer

The functions required to implement a Transfer plugin are:

```
# apply the Fourier-space kernel
In [6]: help(Transfer.__call__)
Help on function __call__ in module nbodykit.core.transfer:

__call__(self, pm, complex)
    Apply the transfer function to the complex field

    Parameters
    ----------
    pm : ParticleMesh
        the particle mesh object which holds possibly useful
        information, i.e, `w` or `k` arrays
    complex : array_like
        the complex array to apply the transfer to
```

nbodykit.core package

class `nbodykit.core.`**`Algorithm`**(**args*, ***kwargs*)

Bases: `nbodykit.plugins.PluginBase`

Mount point for plugins which provide an interface for running one of the high-level algorithms, i.e, power spectrum calculation or FOF halo finder

Plugins of this type should provide the following attributes:

plugin_name [str] A class attribute that defines the name of the plugin in the registry

register [classmethod] A class method taking no arguments that updates the `ConstructorSchema` with the arguments needed to initialize the class

run [method] function that will run the algorithm

save [method] save the result of the algorithm computed by `Algorithm.run()`

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	The class method responsible fill the class's schema with the relevant parameters from the _
`from_config`(parsed)	Instantiate a plugin from this extension point,
`run`()	Run the algorithm
`save`(output, result)	Save the results of the algorithm run

`logger` = **<logging.Logger object>**

`run`()

Run the algorithm

Returns **result** : tuple

the tuple of results that will be passed to `Algorithm.save()`

`save`(*output*, *result*)

Save the results of the algorithm run

Parameters **output** : str

the name of the output file to save results too

result : tuple

the tuple of results returned by `Algorithm.run()`

class `nbodykit.core.`**`Source`**(**args*, ***kwargs*)

Bases: `nbodykit.plugins.PluginBase`

A base class to represent an object that combines the processes of reading / generating data and painting to a RealField

Attributes

BoxSize	A 3-vector specifying the size of the box for this source
attrs	
columns	
string	A unique identifier for the plugin, using the id()

Methods

compute(*args, **kwargs)	Our version of dask.compute() that computes
create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	The class method responsible fill the class's schema with the relevant parameters from the _
from_config(parsed)	Instantiate a plugin from this extension point,
paint(pm)	
read(columns)	

BoxSize
A 3-vector specifying the size of the box for this source

attrs

columns

static compute (**args*, ***kwargs*)
Our version of dask.compute() that computes multiple delayed dask collections at once

Parameters **args** : object

Any number of objects. If the object is a dask collection, it's computed and the result is returned. Otherwise it's passed through unchanged.

Notes

The dask default optimizer induces too many (unnecesarry) IO calls – we turn this off feature off by default.

Eventually we want our own optimizer probably.

logger = <logging.Logger object>

paint (*pm*)

read (*columns*)

class nbodykit.core.**DataSource** (**args*, ***kwargs*)
Bases: *nbodykit.core.datasource.DataSourceBase*

Mount point for plugins which refer to the reading of input files. The *read* operation occurs on a DataStream object, which is returned by *open()*.

Default values for any columns to read can be supplied as a dictionary argument to *open()*.

Plugins of this type should provide the following attributes:

plugin_name [str] A class attribute that defines the name of the plugin in the registry

register [classmethod] A class method taking no arguments that updates the ConstructorSchema with the arguments needed to initialize the class

readall: method A method to read all available data at once (uncollectively) and cache the data in memory for repeated calls to *read*

parallel_read: method A method to read data for complex, large data sets. The read operation shall be collective, with each yield generating different sections of the data source on different ranks. No caching of data takes places.

Notes

- a `Cosmology` instance can be passed to any DataSource class via the *cosmo* keyword
- the data will be cached in memory if returned via `readall()`
- the default cache behavior is for the cache to persist while an open DataStream remains, but the cache can be forced to persist via the `DataSource.keep_cache()` context manager

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`MissingColumn`	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	The class method responsible fill the class's schema with the relevant parameters from the _
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
`open`([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
`parallel_read`(columns[, full])	Override this function for complex, large data sets.
`readall`()	Override to provide a method to read all available data at once

exception `MissingColumn`
Bases: `Exception`

`DataSource.keep_cache()`
A context manager that forces the DataSource cache to persist, even if there are no open DataStream objects. This will prevent unwanted and unnecessary re-readings of the DataSource.

The below example details the intended usage. In this example, the data is cached only once, and no re-reading of the data occurs when the second stream is opened.

```
with datasource.keep_cache():
    with datasource.open() as stream1:
        [[pos]] = stream1.read(['Position'], full=True)

    with datasource.open() as stream2:
        [[vel]] = stream2.read(['Velocity'], full=True)
```

`DataSource.logger` = <logging.Logger object>

`DataSource.`**`open`**`(`*defaults={}*`)`

Open the DataSource by returning a DataStream from which the data can be read.

This function also specifies the default values for any columns that are not supported by the DataSource. The defaults are unique to each DataStream, but a DataSource can be opened multiple times (returning different streams) with different default values

Parameters **defaults** : dict, optional

a dictionary providing default values for a given column

Returns **stream** : DataStream

the stream object from which the data can be read via `read()` function

`DataSource.`**`parallel_read`**`(`*columns*, *full=False*`)`

Override this function for complex, large data sets. The read operation shall be collective, each yield generates different sections of the datasource. No caching of data takes places.

If the DataSource does not provide a column in *columns*, *None* should be returned.

Parameters **columns** : list of str

the list of data columns to return

full : bool, optional

if *True*, any *bunchsize* parameters will be ignored, so that each rank will read all of its specified data section at once

Returns **data** : list

a list of the data for each column in columns; if the data source does not provide a given column, that element should be *None*

Notes

- This function will be called if `DataStream.readall()` is not implemented
- The intention is for this function to handle complex and large data sets, where parallel I/O across ranks is required to avoid memory and I/O issues

`DataSource.`**`readall`**`()`

Override to provide a method to read all available data at once (uncollectively) and cache the data in memory for repeated calls to *read*

Returns **data** : dict

a dictionary of all supported data for the data source; keys give the column names and values are numpy arrays

Notes

- By default, `DataStream.read()` calls this function on the root rank to read all available data, and then scatters the data evenly across all available ranks
- The intention is to reduce the complexity of implementing a simple and small data source, for which reading all data at once is feasible

DataSource.**size**

The total size of the DataSource.

The user can set this explicitly (only once per datasource) if the size is known before DataStream.read() is called

class nbodykit.core.**GridSource**(**args*, ***kwargs*)

Bases: *nbodykit.core.datasource.DataSourceBase*

A DataSource reading directly already on a grid

Attributes

string	A unique identifier for the plugin, using the id()

Methods

BoxSizeParser(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	The class method responsible fill the class's schema with the relevant parameters from the _
from_config(parsed)	Instantiate a plugin from this extension point,
read(real)	Read into a real field

logger = <logging.Logger object>

read(*real*)

Read into a real field

class nbodykit.core.**Painter**(*paintbrush*)

Bases: *nbodykit.plugins.PluginBase*

Mount point for plugins which refer to the painting of data, i.e., gridding a field to a mesh

Plugins of this type should provide the following attributes:

plugin_name [str] A class attribute that defines the name of the plugin in the registry

register [classmethod] A class method taking no arguments that updates the ConstructorSchema with the arguments needed to initialize the class

paint [method] A method that performs the painting of the field.

Attributes

string	A unique identifier for the plugin, using the id()

Methods

basepaint(real, position, paintbrush[, weight])	The base function for painting that is used by default.
create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs pas
fill_schema()	The class method responsible fill the class's schema with the relevant paramete

Table 1.10 – continued from previous page

`from_config`(parsed)	Instantiate a plugin from this extension point,
`paint`(pm, datasource)	Paint the DataSource specified to a mesh
`shiftedpaint`(real1, real2, position, paintbrush)	paint to two real fields for interlacing

`__init__`(*paintbrush*)

`basepaint`(*real*, *position*, *paintbrush*, *weight=None*)
The base function for painting that is used by default. This handles the domain decomposition steps that are necessary to complete before painting.

Parameters **pm** : `ParticleMesh`

particle mesh object that does the painting

position : array_like

the position data

paintbrush : string

picking the paintbrush. Available ones are from documentation of pm.RealField.paint().

weight : array_like, optional

the weight value to use when painting

`logger` = **<logging.Logger object>**

`paint`(*pm*, *datasource*)
Paint the DataSource specified to a mesh

Parameters **pm** : `ParticleMesh`

particle mesh object that does the painting

datasource : DataSource

the data source object representing the field to paint onto the mesh

Returns **stats** : dict

dictionary of statistics related to painting and reading of the DataSource

`required_attributes` = **['paintbrush']**

`shiftedpaint`(*real1*, *real2*, *position*, *paintbrush*, *weight=None*, *shift=0.5*)
paint to two real fields for interlacing

class `nbodykit.core.`**`Transfer`**(**args*, ***kwargs*)
Bases: `nbodykit.plugins.PluginBase`

Mount point for plugins which apply a k-space transfer function to the Fourier transfrom of a datasource field

Plugins of this type should provide the following attributes:

plugin_name [str] class attribute that defines the name of the plugin in the registry

register [classmethod] a class method taking no arguments that updates the `ConstructorSchema` with the arguments needed to initialize the class

__call__ [method] function that will apply the transfer function to the complex array

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`__call__`(pm, complex)	Apply the transfer function to the complex field
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	The class method responsible fill the class's schema with the relevant parameters from the _
`from_config`(parsed)	Instantiate a plugin from this extension point,

`logger` **= <logging.Logger object>**

Subpackages

nbodykit.core.algorithms package

class `nbodykit.core.algorithms.`**`Algorithm`**(**args*, ***kwargs*)

Bases: *`nbodykit.plugins.PluginBase`*

Mount point for plugins which provide an interface for running one of the high-level algorithms, i.e, power spectrum calculation or FOF halo finder

Plugins of this type should provide the following attributes:

plugin_name [str] A class attribute that defines the name of the plugin in the registry

register [classmethod] A class method taking no arguments that updates the `ConstructorSchema` with the arguments needed to initialize the class

run [method] function that will run the algorithm

save [method] save the result of the algorithm computed by *`Algorithm.run()`*

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	The class method responsible fill the class's schema with the relevant parameters from the _
`from_config`(parsed)	Instantiate a plugin from this extension point,
`run`()	Run the algorithm
`save`(output, result)	Save the results of the algorithm run

`logger` **= <logging.Logger object>**

`run`()

Run the algorithm

Returns **result** : tuple

the tuple of results that will be passed to `Algorithm.save()`

save(*output*, *result*)

Save the results of the algorithm run

Parameters **output** : str

the name of the output file to save results too

result : tuple

the tuple of results returned by `Algorithm.run()`

Submodules

nbodykit.core.algorithms.BianchiFFTPower module

class nbodykit.core.algorithms.BianchiFFTPower.**BianchiPowerAlgorithm**(*data*, *randoms*, *Nmesh*, *max_ell*, *paintbrush='cic'*, *dk=None*, *kmin=0.0*, *BoxSize=None*, *BoxPad=0.02*, *compute_fkp_weights=False*, *P0_fkp=None*, *nbar=None*, *fsky=None*, *factor_hexadecapole=False*, *keep_cache=False*)

Bases: `nbodykit.core.algorithms.Algorithm`

Algorithm to compute the power spectrum multipoles using FFTs for a data survey with non-trivial geometry

The algorithm used to compute the multipoles is detailed in Bianchi et al. 2015 (http://adsabs.harvard.edu/abs/2015MNRAS.453L..11B)

Notes

The algorithm saves the power spectrum result to a plaintext file, as well as the meta-data associted with the algorithm.

The columns names are:

- **k** [] the mean value for each *k* bin
- **power_X.real, power_X.imag** [multipoles only] the real and imaginary components for the *X* multipole
- **modes** [] the number of Fourier modes averaged together in each bin

The plaintext files also include meta-data associated with the algorithm:

- **Lx, Ly, Lz** [] the length of each side of the box used when computing FFTs

- **volumne** [] the volume of the box; equal to `Lx*Ly*Lz`
- **N_data** [] the number of objects in the "data" catalog
- **N_ran** [] the number of objects in the "randoms" catalog
- **alpha** [] the ratio of data to randoms; equal to `N_data/N_ran`
- **S_data** [] the unnormalized shot noise for the "data" catalog; see equations 13-15 of Beutler et al. 2014
- **S_ran** [] the same as *S_data*, but for the "randoms" catalog
- **A_data** [] the power spectrum normalization, as computed from the "data" catalog; see equations 13-15 of Beutler et al. 2014 for further details
- **A_ran** [] the same as *A_data*, but for the "randoms" catalog; this is the actual value used to normalize the power spectrum, but its value should be very close to *A_data*
- **shot_noise** [] the final shot noise for the monopole; equal to `(S_ran + S_data)/A_ran`

See *`nbodykit.files.Read1DPlainText()`* and *`nbodykit.dataset.Power1dDataSet.from_nbkit()`* for examples on how to read the the plaintext file.

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`run`()	Run the algorithm, which computes and returns the power spectrum
`save`(output, result)	Save the power spectrum results to the specified output file

`__init__`(*data*, *randoms*, *Nmesh*, *max_ell*, *paintbrush='cic'*, *dk=None*, *kmin=0.0*, *BoxSize=None*, *BoxPad=0.02*, *compute_fkp_weights=False*, *P0_fkp=None*, *nbar=None*, *fsky=None*, *factor_hexadecapole=False*, *keep_cache=False*)

power spectrum multipoles using FFTs for a data survey with non-trivial geometry, as detailed in Bianchi et al. 2015 (1505.05341)

Parameters **data** : DataSource.from_config

DataSource representing the *data* catalog

randoms : DataSource.from_config

DataSource representing the *randoms* catalog

Nmesh : int

the number of cells in the gridded mesh (per axis)

max_ell : { '0', '2', '4' }

compute multipoles up to and including this ell value

paintbrush : { 'cic', 'tsc' }, optional the density assignment kernel to use when painting; CIC (2nd order) or TSC (3rd order) (default: cic)

dk : float, optional

the spacing of k bins to use; if not provided, the fundamental mode of the box is used

kmin : float, optional

the edge of the first *k* bin to use; default is 0 (default: 0.0)

BoxSize : BoxSizeParser, optional

the size of the box; if not provided, automatically computed from the *randoms* catalog

BoxPad : float, optional

when setting the box size automatically, apply this additional buffer (default: 0.02)

compute_fkp_weights : bool, optional

if set, use FKP weights, computed from *P0_fkp* and the provided *nbar* (default: False)

P0_fkp : float, optional

the fiducial power value *P0* used to compute FKP weights

nbar : str, optional

read *nbar(z)* from this file, which provides two columns (z, nbar)

fsky : float, optional

the sky area fraction of the tracer catalog, used in the volume calculation of *nbar*

factor_hexadecapole : bool, optional

use the factored expression for the hexadecapole (ell=4) from eq. 27 of Scoccimarro 2015 (1506.02729) (default: False)

keep_cache : bool, optional

if *True*, force the data cache to persist while the algorithm instance is valid (default: False)

classmethod `fill_schema`()

`logger` = **<logging.Logger object>**

`plugin_name` = **'BianchiFFTPower'**

`run`()

Run the algorithm, which computes and returns the power spectrum

`save`(*output*, *result*)

Save the power spectrum results to the specified output file

Parameters **output** : str

the string specifying the file to save

result : tuple

the tuple returned by *run()* – first argument specifies the bin edges and the second is a dictionary holding the data results

`schema` = **<ConstructorSchema: 16 parameters (12 optional)>**

nbodykit.core.algorithms.ExampleAlgorithm module

class `nbodykit.core.algorithms.ExampleAlgorithm.`**`Describe`**(*datasource*, *column='Position'*)

Bases: *`nbodykit.core.algorithms.Algorithm`*

A simple example Algorithm that loads a specific column from a DataSource and prints the min/max of the column

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`run`()	Run the algorithm, which does nothing
`save`(output, data)	

`__init__`(*datasource*, *column='Position'*)

describe a specific column of the input DataSource

Parameters datasource : DataSource.from_config

the DataSource to describe; run *nbkit.py –list-datasources* for all options

column : str, optional

the column in the DataSource to describe (default: Position)

classmethod `fill_schema`()

`logger` = **<logging.Logger object>**

`plugin_name` = **'Describe'**

`run`()

Run the algorithm, which does nothing

`save`(*output*, *data*)

`schema` = **<ConstructorSchema: 3 parameters (2 optional)>**

class `nbodykit.core.algorithms.ExampleAlgorithm.`**`Play`**(*source*, *columns=None*)

Bases: *`nbodykit.core.algorithms.Algorithm`*

A simple example Algorithm that plays the 'read' interface of a Source and does min/max on selected columns

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	
from_config(parsed)	Instantiate a plugin from this extension point,
run()	Run the algorithm, which does nothing
save(output, result)	

__init__(*source*, *columns=None*)
describe a specific column of the input DataSource

Parameters **source** : Source.from_config

the DataSource to describe; run *nbkit.py –list-datasources* for all options

columns : str, optional

the column in the DataSource to describe

classmethod **fill_schema**()

logger = <logging.Logger object>

plugin_name = 'Play'

run()
Run the algorithm, which does nothing

save(*output*, *result*)

schema = <ConstructorSchema: 3 parameters (2 optional)>

nbodykit.core.algorithms.FOF module

class nbodykit.core.algorithms.FOF.**FOFAlgorithm**(*datasource*, *linklength*, *absolute=False*, *without_labels=False*, *nmin=32*, *calculate_initial_position=False*)

Bases: *nbodykit.core.algorithms.Algorithm*

Attributes

string	A unique identifier for the plugin, using the id()

Methods

create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	
from_config(parsed)	Instantiate a plugin from this extension point,
run()	
save(output, data)	

__init__(*datasource*, *linklength*, *absolute=False*, *without_labels=False*, *nmin=32*, *calculate_initial_position=False*)
a Friends-of-Friends (FOF) halo finder

Parameters **datasource** : DataSource.from_config

DataSource objects to run FOF against; run *nbkit.py –list-datasources* for all options

linklength : float

the linking length in either absolute or relative units

absolute : bool, optional

If set, the linking length is in absolute units, otherwise it is relative to the mean particle separation; default is *False* (default: False)

without_labels : bool, optional

do not store labels (default: False)

nmin : int, optional

minimum number of particles in a halo (default: 32)

calculate_initial_position : bool, optional

If set, calculate the initial position of halos based on the InitialPosition field of DataSource (default: False)

classmethod `fill_schema`()

`logger` = **<logging.Logger object>**

`plugin_name` = **'FOF'**

`run`()

`save`(*output*, *data*)

`schema` = **<ConstructorSchema: 7 parameters (5 optional)>**

nbodykit.core.algorithms.FOF6D module

class `nbodykit.core.algorithms.FOF6D.`**`FOF6DAlgorithm`**(*datasource*, *halolabel*, *linklength=0.078*, *vfactor=0.368*, *nmin=32*)

Bases: *`nbodykit.core.algorithms.Algorithm`*

An algorithm to find subhalos from FOF groups; a variant of FOF6D

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`run`()	Run the FOF6D Algorithm
`save`(output, data)	Save the result

`__init__`(*datasource*, *halolabel*, *linklength=0.078*, *vfactor=0.368*, *nmin=32*)
finding subhalos from FOF groups; a variant of FOF6D

Parameters **datasource** : DataSource.from_config

DataSource objects to run FOF against; run *nbkit.py –list-datasources* for all options

halolabel : DataSource.from_config

data source for the halo label files; column name is Label

linklength : float, optional

the linking length (default: 0.078)

vfactor : float, optional

velocity linking length in units of 1d velocity dispersion. (default: 0.368)

nmin : int, optional

minimum number of particles in a halo (default: 32)

classmethod `fill_schema()`

`logger` **= <logging.Logger object>**

`plugin_name` **= 'FOF6D'**

`run()`

Run the FOF6D Algorithm

`save(output, data)`

Save the result

`schema` **= <ConstructorSchema: 6 parameters (4 optional)>**

`nbodykit.core.algorithms.FOF6D.so(center, data, r1, nbar, thresh=200)`

`nbodykit.core.algorithms.FOF6D.subfof(pos, vel, ll, vfactor, haloid, Ntot, boxsize)`

nbodykit.core.algorithms.FiberCollisions module

class `nbodykit.core.algorithms.FiberCollisions.FiberCollisionsAlgorithm(datasource, collision_radius=0.0172222222222, seed=None)`

Bases: `nbodykit.core.algorithms.Algorithm`

Run an angular FOF algorithm to determine fiber collision groups from an input catalog, and then assign fibers such that the maximum amount of object receive a fiber. This amounts to determining the following population of objects:

- **population 1:** the maximal "clean" sample of objects in which each object is not angularly collided with any other object in this subsample
- **population 2:** the potentially-collided objects; these objects are those that are fiber collided + those that have been "resolved" due to multiple coverage in tile overlap regions

See Guo et al. 2010 (http://arxiv.org/abs/1111.6598)for further details

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	
from_config(parsed)	Instantiate a plugin from this extension point,
run()	Compute the FOF collision groups and assign fibers, such that
save(output, result)	Write the *Label*, *Collided*, and *NeighborID* arrays

__init__(*datasource*, *collision_radius=0.017222222222222226*, *seed=None*)
the application of fiber collisions to a galaxy survey

Parameters **datasource** : RaDecDataSource

RaDecRedshift DataSource; run 'nbkit.py –list-datasources RaDecRedshift for details

collision_radius : float, optional

the size of the angular collision radius (in degrees) (default: 0.017222222222222226)

seed : int, optional

seed the random number generator explicitly, for reproducibility

classmethod fill_schema()

logger = <logging.Logger object>

plugin_name = 'FiberCollisions'

run()
Compute the FOF collision groups and assign fibers, such that the maximum number of objects receive fibers

Returns result: array_like

a structured array with 3 fields:

Label [] the group labels for each object in the input DataSource; label == 0 objects are not in a group

Collided [] a flag array specifying which objects are collided, i.e., do not receive a fiber

NeighborID [] for those objects that are collided, this gives the (global) index of the nearest neighbor on the sky (0-indexed), else it is set to -1

save(*output*, *result*)
Write the *Label*, *Collided*, and *NeighborID* arrays as a Pandas DataFrame to an HDF file, with key *FiberCollisonGroups*

schema = <ConstructorSchema: 4 parameters (3 optional)>

nbodykit.core.algorithms.FiberCollisions.**RaDecDataSource**(*d*)

nbodykit.core.algorithms.PaintGrid module

class nbodykit.core.algorithms.PaintGrid.**PaintGridAlgorithm**(*Nmesh*, *DataSource*, *Painter=None*, *paintbrush='cic'*, *paintNmesh=None*, *dataset='PaintGrid'*, *Nfile=0*, *writeFourier=False*)

Bases: nbodykit.core.algorithms.Algorithm

Algorithm to paint a data source to a 3D configuration space grid.

Notes

The algorithm saves the grid to a bigfile File.

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`run`()	Run the algorithm, which computes and returns the grid in C_CONTIGUOUS order partitio
`save`(output, result)	Save the power spectrum results to the specified output file

`__init__`(*Nmesh*, *DataSource*, *Painter=None*, *paintbrush='cic'*, *paintNmesh=None*, *dataset='PaintGrid'*, *Nfile=0*, *writeFourier=False*)

periodic power spectrum calculator via FFT

Parameters **Nmesh** : int

the number of cells in the gridded mesh

DataSource : DataSource.from_config, GridSource.from_config

DataSource)

Painter : Painter.from_config, optional

the Painter; run *nbkit.py –list-painters* for all options

paintbrush : { 'cic', 'tsc' }, optional

the density assignment kernel to use when painting; CIC (2nd order) or TSC (3rd order) (default: cic)

paintNmesh : int, optional

The painting Nmesh. The grid will be Fourier resampled to Nmesh before output. A value larger than Nmesh can reduce grid artifacts.

dataset : optional

name of dataset to write to (default: PaintGrid)

Nfile : optional

number of files (default: 0)

writeFourier : bool, optional

Write complex Fourier modes instead? (default: False)

classmethod `fill_schema`()

`logger` **= <logging.Logger object>**

`plugin_name` **= 'PaintGrid'**

`run`()
Run the algorithm, which computes and returns the grid in C_CONTIGUOUS order partitioned by ranks.

`save`(*output*, *result*)
Save the power spectrum results to the specified output file

Parameters **output** : str

the string specifying the file to save

result : tuple

the tuple returned by *run()* – first argument specifies the bin edges and the second is a dictionary holding the data results

`schema` **= <ConstructorSchema: 9 parameters (7 optional)>**

nbodykit.core.algorithms.PairCountCorrelation module

class `nbodykit.core.algorithms.PairCountCorrelation.`**`PairCountCorrelationAlgorithm`**(*mode*, *rbins*, *field*, *other=None*, *subsample=1*, *los='z'*, *Nmu=10*, *poles=[]*)

Bases: *`nbodykit.core.algorithms.Algorithm`*

Algorithm to compute the 1d or 2d correlation function and/or multipoles via direct pair counting

Notes

The algorithm saves the correlation function result to a plaintext file, as well as the meta-data associted with the algorithm. The names of the columns saved to file are:

- **r** [] the mean separation in each *r* bin
- **mu** [2D corr only] the mean value for each *mu* bin
- **corr** [] the correlation function value
- **corr_X :** the *X* multipole of the correlation function
- **RR** [] the number of random-random pairs in each bin; used to properly normalize the correlation function
- **N** [] the number of pairs averaged over in each bin to compute the correlation function

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`run`()	Run the pair-count correlation function and return the result
`save`(output, result)	Save the result returned by *run()* to the filename specified by *output*

`__init__`(*mode*, *rbins*, *field*, *other=None*, *subsample=1*, *los='z'*, *Nmu=10*, *poles=[]*)
correlation function calculator via pair counting

Parameters **mode** : { '1d', '2d' }

measure the correlation function in *1d* or *2d*

rbins : binning_type

the string specifying the binning to use

field : DataSource.from_config

the first *DataSource* of objects to correlate; run *nbkit.py –list-datasources* for all options

other : DataSource.from_config, optional

the other *DataSource* of objects to cross-correlate with; run *nbkit.py –list-datasources* for all options

subsample : int, optional

use 1 out of every N points (default: 1)

los : { 'x', 'y', 'z' }, optional

the line-of-sight: the angle *mu* is defined with respect to (default: z)

Nmu : int, optional

if *mode == 2d*, the number of mu bins covering mu=[-1,1] (default: 10)

poles : int, optional

compute the multipoles for these *ell* values from xi(r,mu) (default: [])

classmethod `fill_schema`()

`logger` = **<logging.Logger object>**

`plugin_name` = **'PairCountCorrelation'**

`run`()
Run the pair-count correlation function and return the result

Returns **edges** : list or array_like

the array of 1d bin edges or a list of the bin edges in each dimension

result : dict

dictionary holding the data results (with associated names as keys) – this results *corr*, *RR*, *N* + the mean bin values

`save`(*output*, *result*)
Save the result returned by *run()* to the filename specified by *output*

Parameters **output** : str

the string specifying the file to save

result : tuple

the tuple returned by *run()* – first argument specifies the bin edges and the second is a dictionary holding the data results

schema = <ConstructorSchema: 9 parameters (6 optional)>

nbodykit.core.algorithms.PairCountCorrelation.**binning_type**(*s*)

Type conversion for use on the command-line that converts a string to an array of bin edges

nbodykit.core.algorithms.PeriodicBox module

class nbodykit.core.algorithms.PeriodicBox.**FFTCorrelationAlgorithm**(*mode*, *Nmesh*, *field*, *other=None*, *los='z'*, *Nmu=5*, *dk=None*, *kmin=0.0*, *quiet=False*, *poles=[]*, *paintbrush='cic'*)

Bases: *nbodykit.core.algorithms.Algorithm*

Algorithm to compute the 1d or 2d correlation function and/or multipoles in a periodic box

The algorithm simply takes the Fast Fourier Transform (FFT) of the measured power spectrum, as computed by *FFTPowerAlgorithm*, to compute the correlation function

Notes

The algorithm saves the correlation function result to a plaintext file, as well as the meta-data associted with the algorithm. The names of the columns saved to file are:

- **r** [] the mean separation in each *r* bin
- **mu** [2D corr only] the mean value for each *mu* bin
- **corr** [] the correlation function value
- **corr_X :** the *X* multipole of the correlation function
- **modes** [] the number of Fourier modes averaged together in each bin

The plaintext files also include meta-data associated with the algorithm:

- **Lx, Ly, Lz** [] the length of each side of the box used when computing FFTs
- **volumne** [] the volume of the box; equal to `Lx*Ly*Lz`
- **N1** [] the number of objects in the 1st catalog
- **N2** [] the number of objects in the 2nd catalog; equal to *N1* if the power spectrum is an auto spectrum

See *nbodykit.files.Read1DPlainText()*, *nbodykit.files.Read2DPlainText()* and *nbodykit.dataset.Corr1dDataSet.from_nbkit()* *nbodykit.dataset.Corr2dDataSet.from_nbkit()* for examples on how to read the the plaintext file.

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`run`()	Run the algorithm, which computes and returns the correlation function
`save`(output, result)	Save the correlation function results to the specified output file

`__init__`(*mode*, *Nmesh*, *field*, *other=None*, *los='z'*, *Nmu=5*, *dk=None*, *kmin=0.0*, *quiet=False*, *poles=[]*, *paintbrush='cic'*)

correlation spectrum calculator via FFT in a periodic box

Parameters **mode** : { '1d', '2d' }

compute the power as a function of *k* or *k* and *mu*

Nmesh : int

the number of cells in the gridded mesh

field : FieldType

first data field; a tuple of (DataSource, Painter, Transfer) The 3 subfields are:

DataSource [DataSource.from_config, GridSource.from_config] the 1st DataSource; run *nbkit.py –list-datasources* for all options

Painter [Painter.from_config] the 1st Painter; run *nbkit.py –list-painters* for all options

Transfer [Transfer.from_config] the 1st Transfer chain; run *nbkit.py –list-transfers* for all options

other : FieldType, optional

the other data field; a tuple of (DataSource, Painter, Transfer) The 3 subfields are:

DataSource [DataSource.from_config, GridSource.from_config] the 2nd DataSource; run *nbkit.py –list-datasources* for all options

Painter [Painter.from_config] the 2nd Painter; run *nbkit.py –list-painters* for all options

Transfer [Transfer.from_config] the 2nd Transfer chain; run *nbkit.py –list-transfers* for all options

los : { 'x', 'y', 'z' }, optional

the line-of-sight direction – the angle *mu* is defined with respect to (default: z)

Nmu : int, optional

the number of mu bins to use from mu=[0,1]; if *mode = 1d*, then *Nmu* is set to 1 (default: 5)

dk : float, optional

the spacing of k bins to use; if not provided, the fundamental mode of the box is used

kmin : float, optional

the edge of the first *k* bin to use; default is 0 (default: 0.0)

quiet : bool, optional

silence the logging output (default: False)

poles : int, optional

if specified, also compute these multipoles from P(k,mu) (default: [])

paintbrush : { 'cic', 'tsc' }, optional

the density assignment kernel to use when painting; CIC (2nd order) or TSC (3rd order) (default: cic)

comm : optional

the global MPI communicator

classmethod `fill_schema`()

`logger` = <logging.Logger object>

`plugin_name` = 'FFTCorrelation'

`run`()

Run the algorithm, which computes and returns the correlation function

`save`(*output*, *result*)

Save the correlation function results to the specified output file

Parameters **output** : str

the string specifying the file to save

result : tuple

the tuple returned by *run()* – first argument specifies the bin edges and the second is a dictionary holding the data results

`schema` = <ConstructorSchema: 12 parameters (9 optional)>

class `nbodykit.core.algorithms.PeriodicBox.`**`FFTPowerAlgorithm`**(*mode*, *Nmesh*, *field*, *other=None*, *los='z'*, *Nmu=5*, *dk=None*, *kmin=0.0*, *quiet=False*, *poles=[]*, *paintbrush='cic'*)

Bases: *`nbodykit.core.algorithms.Algorithm`*

Algorithm to compute the 1d or 2d power spectrum and/or multipoles in a periodic box, using a Fast Fourier Transform (FFT)

Notes

The algorithm saves the power spectrum results to a plaintext file, as well as the meta-data associted with the algorithm. The names of the columns saved to file are:

- **k** [] the mean value for each *k* bin
- **mu** [2D power only] the mean value for each *mu* bin

- **power.real, power.imag** [1D/2D power only] the real and imaginary components of 1D power
- **power_X.real, power_X.imag** [multipoles only] the real and imaginary components for the *X* multipole
- **modes** [] the number of Fourier modes averaged together in each bin

The plaintext files also include meta-data associated with the algorithm:

- **Lx, Ly, Lz** [] the length of each side of the box used when computing FFTs
- **volumne** [] the volume of the box; equal to `Lx*Ly*Lz`
- **N1** [] the number of objects in the 1st catalog
- **N2** [] the number of objects in the 2nd catalog; equal to *N1* if the power spectrum is an auto spectrum

See `nbodykit.files.Read1DPlainText()`, `nbodykit.files.Read2DPlainText()` and `nbodykit.dataset.Power1dDataSet.from_nbkit()` `nbodykit.dataset.Power2dDataSet.from_nbkit`
for examples on how to read the the plaintext file.

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`run`()	Run the algorithm, which computes and returns the power spectrum
`save`(output, result)	Save the power spectrum results to the specified output file

`__init__`(*mode, Nmesh, field, other=None, los='z', Nmu=5, dk=None, kmin=0.0, quiet=False, poles=[], paintbrush='cic'*)

periodic power spectrum calculator via FFT

Parameters **mode** : { '1d', '2d' }

compute the power as a function of *k* or *k* and *mu*

Nmesh : int

the number of cells in the gridded mesh

field : FieldType

first data field; a tuple of (DataSource, Painter, Transfer) The 3 subfields are:

DataSource [DataSource.from_config, GridSource.from_config] the 1st DataSource; run *nbkit.py –list-datasources* for all options

Painter [Painter.from_config] the 1st Painter; run *nbkit.py –list-painters* for all options

Transfer [Transfer.from_config] the 1st Transfer chain; run *nbkit.py –list-transfers* for all options

other : FieldType, optional

the other data field; a tuple of (DataSource, Painter, Transfer) The 3 subfields are:

DataSource [DataSource.from_config, GridSource.from_config] the 2nd DataSource; run *nbkit.py –list-datasources* for all options

Painter [Painter.from_config] the 2nd Painter; run *nbkit.py –list-painters* for all options

Transfer [Transfer.from_config] the 2nd Transfer chain; run *nbkit.py –list-transfers* for all options

los : { 'x', 'y', 'z' }, optional

the line-of-sight direction – the angle *mu* is defined with respect to (default: z)

Nmu : int, optional

the number of mu bins to use from mu=[0,1]; if *mode = 1d*, then *Nmu* is set to 1 (default: 5)

dk : float, optional

the spacing of k bins to use; if not provided, the fundamental mode of the box is used

kmin : float, optional

the edge of the first *k* bin to use; default is 0 (default: 0.0)

quiet : bool, optional

silence the logging output (default: False)

poles : int, optional

if specified, also compute these multipoles from P(k,mu) (default: [])

paintbrush : { 'cic', 'tsc' }, optional

the density assignment kernel to use when painting; CIC (2nd order) or TSC (3rd order) (default: cic)

classmethod `fill_schema()`

`logger` **= <logging.Logger object>**

`plugin_name` **= 'FFTPower'**

`run()`

Run the algorithm, which computes and returns the power spectrum

`save`(*output*, *result*)

Save the power spectrum results to the specified output file

Parameters **output** : str

the string specifying the file to save

result : tuple

the tuple returned by *run()* – first argument specifies the bin edges and the second is a dictionary holding the data results

`schema` **= <ConstructorSchema: 12 parameters (9 optional)>**

`nbodykit.core.algorithms.PeriodicBox.FieldType`(*ns*)

Construct and return a *Field*: a tuple of (*DataSource*, *Painter*, *Transfer*)

Parameters **fields_dict** : OrderedDict an ordered dictionary where the keys are Plugin names and the values are instantiated Plugins

Returns **field** : list

list of (DataSource, Painter, Transfer)

Notes

•the default Painter is set to *DefaultPainter*

•the default Transfer chain is set to [*NormalizeDC*, *RemoveDC*, *AnisotropicCIC*]

nbodykit.core.algorithms.RedshiftHistogram module

`nbodykit.core.algorithms.RedshiftHistogram.`**`RedshiftBins`**(*bins*)

The redshift bins to use when computing n(z)

Parameters **bins** : int or list of scalars

If *bins* is a integer, it defines the number of equal-width bins in the given range. If *bins* is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths

class `nbodykit.core.algorithms.RedshiftHistogram.`**`RedshiftHistogramAlgorithm`**(*datasource*, *bins=None*, *fsky=1.0*, *weight_col='Weight'*)

Bases: *`nbodykit.core.algorithms.Algorithm`*

Algorithm to compute the mean number density of as a function of redshift `n(z)` for the input `DataSource`

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`run`()	Run the algorithm, which returns (z, n(z))
`save`(output, result)	Write (z_cen, n(z_cen)) to the specified file

`__init__`(*datasource*, *bins=None*, *fsky=1.0*, *weight_col='Weight'*)

compute n(z) from the input DataSource

Parameters **datasource** : DataSource.from_config

DataSource with a *Redshift* column to compute n(z) from

bins : RedshiftBins, optional

the input redshift bins, specified as either as an integer or sequence of floats

fsky : float, optional

the sky area fraction, used in the volume calculation for *n(z)* (default: 1.0)

weight_col : str, optional

the name of the column to use as a weight (default: Weight)

classmethod `fill_schema()`

`logger` **= <logging.Logger object>**

`plugin_name` **= 'RedshiftHistogram'**

`run()`

Run the algorithm, which returns (z, n(z))

Returns **zbins** : array_like

the redshift bin edges

z_cen : array_like

the center value of each redshift bin

nz : array_like

the n(z_cen) value

`save`(*output*, *result*)

Write (z_cen, n(z_cen)) to the specified file

Parameters **output** : str

the string specifying the file to save

result : tuple

the tuple returned by *run()*, which holds (z, nz)

`schema` **= <ConstructorSchema: 5 parameters (4 optional)>**

`nbodykit.core.algorithms.RedshiftHistogram.scotts_bin_width`(*data*)

Return the optimal histogram bin width using Scott's rule

nbodykit.core.algorithms.Subsample module

class `nbodykit.core.algorithms.Subsample.Subsample`(*datasource*, *Nmesh*, *seed=12345*, *ratio=0.01*, *smoothing=None*, *format='hdf5'*)

Bases: *nbodykit.core.algorithms.Algorithm*

Algorithm to create a subsample from a DataSource, and evaluate the density (1 + delta), smoothed at the given scale

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs pa

Continu

Table 1.38 – continued from previous page

fill_schema()	
from_config(parsed)	Instantiate a plugin from this extension point,
run()	Run the Subsample algorithm
save(output, data)	
write_hdf5(subsample, output)	
write_mwhite_subsample(subsample, output)	

`__init__`(*datasource*, *Nmesh*, *seed=12345*, *ratio=0.01*, *smoothing=None*, *format='hdf5'*)
create a subsample from a DataSource, and evaluate density (1 + delta) smoothed at the given scale

Parameters **datasource** : DataSource.from_config

the DataSource to read; run *nbkit.py –list-datasources* for all options

Nmesh : int

the size of FFT mesh for painting

seed : int, optional

the random seed (default: 12345)

ratio : float, optional

the fraction of particles to keep (default: 0.01)

smoothing : float, optional

the smoothing length in distance units. It has to be greater than the mesh resolution. Otherwise the code will die. Default is the mesh resolution.

format : { 'hdf5', 'mwhite' }, optional

the format of the output (default: hdf5)

classmethod `fill_schema`()

`logger` = **<logging.Logger object>**

`plugin_name` = **'Subsample'**

`run`()
Run the Subsample algorithm

`save`(*output*, *data*)

`schema` = **<ConstructorSchema: 7 parameters (5 optional)>**

`write_hdf5`(*subsample*, *output*)

`write_mwhite_subsample`(*subsample*, *output*)

nbodykit.core.algorithms.TestBoxSize module

class `nbodykit.core.algorithms.TestBoxSize.`**`TestBoxSizeAlgorithm`**(*datasource*, *BoxSize*)

Bases: *nbodykit.core.algorithms.Algorithm*

A utility algorithm to load a *DataSource* and test if all particles are within the specified BoxSize

Attributes

string	A unique identifier for the plugin, using the id()

Methods

create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	
from_config(parsed)	Instantiate a plugin from this extension point,
run()	Run the algorithm, which reads the position coordinates
save(output, result)	Write out the out-of-bounds particles (done by root)

`__init__`(*datasource*, *BoxSize*)
test if all objects in a DataSource fit within a specified BoxSize

Parameters **datasource** : DataSource.from_config

DataSource holding the positions to test

BoxSize : BoxSizeParser

the size of the box

classmethod `fill_schema`()

`logger` **= <logging.Logger object>**

`plugin_name` **= 'TestBoxSize'**

`run`()
Run the algorithm, which reads the position coordinates from the DataSource and finds any out-of-bounds particles

`save`(*output*, *result*)
Write out the out-of-bounds particles (done by root)

`schema` **= <ConstructorSchema: 3 parameters (1 optional)>**

nbodykit.core.algorithms.TidalTensor module

class `nbodykit.core.algorithms.TidalTensor.`**`TidalTensor`**(*field*, *points*, *Nmesh*, *smoothing=None*)
Bases: *nbodykit.core.algorithms.Algorithm*

Compute and save the tidal force tensor

Attributes

string	A unique identifier for the plugin, using the id()

Methods

NormalizeDC(pm, complex)	removes the DC amplitude. This effectively
Smoothing(pm, complex)	
TidalTensor(u, v)	

Continued on next pa

Table 1.42 – continued from previous page

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`run`()	Run the TidalTensor Algorithm
`save`(output, data)	
`write_hdf5`(data, output)	

`NormalizeDC`(*pm*, *complex*)
removes the DC amplitude. This effectively divides by the mean

`Smoothing`(*pm*, *complex*)

`TidalTensor`(*u*, *v*)

`__init__`(*field*, *points*, *Nmesh*, *smoothing=None*)
compute the tidal force tensor

Parameters **field** : DataSource.from_config

DataSource; run *nbkit.py –list-datasources* for all options

points : DataSource.from_config

A small set of points to calculate tidal force on; run *nbkit.py –list-datasources* for all options

Nmesh : int

Size of FFT mesh for painting

smoothing : float, optional

Smoothing Length in distance units. It has to be greater than the mesh resolution. Otherwise the code will die. Default is the mesh resolution.

classmethod `fill_schema`()

`logger` = <logging.Logger object>

`plugin_name` = 'TidalTensor'

`run`()
Run the TidalTensor Algorithm

`save`(*output*, *data*)

`schema` = <ConstructorSchema: 5 parameters (2 optional)>

`write_hdf5`(*data*, *output*)

nbodykit.core.algorithms.TraceHalo module

class `nbodykit.core.algorithms.TraceHalo.`**`TraceHaloAlgorithm`**(*dest*, *source*, *sourcelabel*)
Bases: *`nbodykit.core.algorithms.Algorithm`*

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`run`()	Run the TraceHalo Algorithm
`save`(output, data)	Save the result

`__init__`(*dest*, *source*, *sourcelabel*)
calculate the halo property based on a different set of halo labels.

Parameters **dest** : DataSource.from_config

type: DataSource

source : DataSource.from_config

type: DataSource

sourcelabel : DataSource.from_config

DataSource of the source halo label files, the Label column is used.

classmethod `fill_schema`()

`logger` = **<logging.Logger object>**

`plugin_name` = **'TraceHalo'**

`run`()
Run the TraceHalo Algorithm

`save`(*output*, *data*)
Save the result

`schema` = **<ConstructorSchema: 4 parameters (1 optional)>**

nbodykit.core.datasource package

class `nbodykit.core.datasource.`**`DataCache`**(*columns*, *data*, *local_size*, *total_size*)
Bases: `object`

A class to cache data in a manner that can be weakly referenced via *weakref*

Attributes

`empty`	Return whether or not the cache is empty

`__init__`(*columns*, *data*, *local_size*, *total_size*)

Parameters **columns** : list of str

the list of columns in the cache

data : list of array_like

a list of data for each column

local_size : int

the size of the data stored locally on this rank

total_size : int

the global size of the data

empty

Return whether or not the cache is empty

class nbodykit.core.datasource.**DataSource**(**args*, ***kwargs*)

Bases: *nbodykit.core.datasource.DataSourceBase*

Mount point for plugins which refer to the reading of input files. The *read* operation occurs on a *DataStream* object, which is returned by *open()*.

Default values for any columns to read can be supplied as a dictionary argument to *open()*.

Plugins of this type should provide the following attributes:

plugin_name [str] A class attribute that defines the name of the plugin in the registry

register [classmethod] A class method taking no arguments that updates the ConstructorSchema with the arguments needed to initialize the class

readall: method A method to read all available data at once (uncollectively) and cache the data in memory for repeated calls to *read*

parallel_read: method A method to read data for complex, large data sets. The read operation shall be collective, with each yield generating different sections of the data source on different ranks. No caching of data takes places.

Notes

- a *Cosmology* instance can be passed to any DataSource class via the *cosmo* keyword
- the data will be cached in memory if returned via readall()
- the default cache behavior is for the cache to persist while an open DataStream remains, but the cache can be forced to persist via the *DataSource.keep_cache()* context manager

Attributes

size	The total size of the DataSource.
string	A unique identifier for the plugin, using the id()

Methods

BoxSizeParser(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
MissingColumn	
create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	The class method responsible fill the class's schema with the relevant parameters from the _
from_config(parsed)	Instantiate a plugin from this extension point,
keep_cache()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
open([defaults])	Open the DataSource by returning a DataStream from which the data can be read.

Co

Table 1.47 – continued from previous page

parallel_read(columns[, full])	Override this function for complex, large data sets.
readall()	Override to provide a method to read all available data at once

exception **MissingColumn**
Bases: Exception

DataSource.**keep_cache**()
A context manager that forces the DataSource cache to persist, even if there are no open DataStream objects. This will prevent unwanted and unnecessary re-readings of the DataSource.

The below example details the intended usage. In this example, the data is cached only once, and no re-reading of the data occurs when the second stream is opened.

```
with datasource.keep_cache():
    with datasource.open() as stream1:
        [[pos]] = stream1.read(['Position'], full=True)

    with datasource.open() as stream2:
        [[vel]] = stream2.read(['Velocity'], full=True)
```

DataSource.**logger = <logging.Logger object>**

DataSource.**open**(*defaults={}*)
Open the DataSource by returning a DataStream from which the data can be read.

This function also specifies the default values for any columns that are not supported by the DataSource. The defaults are unique to each DataStream, but a DataSource can be opened multiple times (returning different streams) with different default values

Parameters **defaults** : dict, optional

a dictionary providing default values for a given column

Returns **stream** : DataStream

the stream object from which the data can be read via read() function

DataSource.**parallel_read**(*columns*, *full=False*)
Override this function for complex, large data sets. The read operation shall be collective, each yield generates different sections of the datasource. No caching of data takes places.

If the DataSource does not provide a column in *columns*, *None* should be returned.

Parameters **columns** : list of str

the list of data columns to return

full : bool, optional

if *True*, any *bunchsize* parameters will be ignored, so that each rank will read all of its specified data section at once

Returns **data** : list

a list of the data for each column in columns; if the data source does not provide a given column, that element should be *None*

Notes

- This function will be called if DataStream.readall() is not implemented

- The intention is for this function to handle complex and large data sets, where parallel I/O across ranks is required to avoid memory and I/O issues

`DataSource.`**`readall`**`()`

Override to provide a method to read all available data at once (uncollectively) and cache the data in memory for repeated calls to *read*

Returns data : dict

a dictionary of all supported data for the data source; keys give the column names and values are numpy arrays

Notes

- By default, *`DataStream.read()`* calls this function on the root rank to read all available data, and then scatters the data evenly across all available ranks
- The intention is to reduce the complexity of implementing a simple and small data source, for which reading all data at once is feasible

`DataSource.`**`size`**

The total size of the DataSource.

The user can set this explicitly (only once per datasource) if the size is known before *`DataStream.read()`* is called

class `nbodykit.core.datasource.`**`DataSourceBase`**`(*args, **kwargs)`

Bases: *`nbodykit.plugins.PluginBase`*

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	The class method responsible fill the class's schema with the relevant parameters from the _
`from_config`(parsed)	Instantiate a plugin from this extension point,

static **`BoxSizeParser`**`(value)`

Read the *BoxSize*, enforcing that the BoxSize must be a scalar or 3-vector

Returns BoxSize : array_like

an array of size 3 giving the box size in each dimension

`logger` **= <logging.Logger object>**

class `nbodykit.core.datasource.`**`DataStream`**`(data, defaults={})`

Bases: `object`

A class to represent an open *`DataSource`* stream.

The class behaves similar to the built-in `open()` for `file` objects. The stream is returned by calling `DataSource.open()` and the class is written such that it can be used with or without the *with* statement, similar to the file `open()` function.

Attributes

`read`(columns[, full])	Read the data corresponding to *columns*

nread	(int) during each iteration of *read*, this will store the number of rows read from the data source

Methods

`close`()	Close the DataStream; no *read* operations are
`isdefault`(name, data)	Return *True* if the input data is equal to the default
`read`(columns[, full])	Read the data corresponding to *columns*

`__init__`(*data*, *defaults={}*)

Parameters **data** : DataSource

the DataSource that this stream returns data from

defaults : dict

dictionary of default values for the specified columns

`close`()

Close the DataStream; no *read* operations are allowed on closed streams

`closed`

Return whether or not the DataStream is closed

`isdefault`(*name*, *data*)

Return *True* if the input data is equal to the default value for this stream

`read`(*columns*, *full=False*)

Read the data corresponding to *columns*

Parameters **columns** : list

list of strings giving the names of the desired columns

full : bool, optional

if *True*, any *bunchsize* parameters will be ignored, so that each rank will read all of its specified data section at once; default is *False*

Returns **data** : list

a list of the data for each column in *columns*

class `nbodykit.core.datasource.`**`GridSource`**(**args*, ***kwargs*)

Bases: `nbodykit.core.datasource.DataSourceBase`

A DataSource reading directly already on a grid

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	The class method responsible fill the class's schema with the relevant parameters from the _
`from_config`(parsed)	Instantiate a plugin from this extension point,
`read`(real)	Read into a real field

`logger` = **<logging.Logger object>**

`read`(*real*)
Read into a real field

Submodules

nbodykit.core.datasource.BigFileGrid module

class `nbodykit.core.datasource.BigFileGrid.`**`BigFileGridSource`**(*path*, *dataset*)
Bases: *`nbodykit.core.datasource.GridSource`*

Class to read field gridded data from a binary file

Notes

- Reading is designed to be done by *GridPainter*, which reads gridded quantity straight into the *ParticleMesh*

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`read`(real)	

`__init__`(*path*, *dataset*)
read gridded field data from a binary file. Fourier resampling is applied if necessary.

Parameters **path** : str

the file path to load the data from

dataset : str

the file path to load the data from

classmethod `fill_schema`()

`logger` = <logging.Logger object>

`plugin_name` = 'BigFileGrid'

`read`(*real*)

`schema` = <ConstructorSchema: 4 parameters (2 optional)>

nbodykit.core.datasource.FOFGroups module

class `nbodykit.core.datasource.FOFGroups.`**`FOFDataSource`**(*path*, *m0*, *dataset='FOFGroups'*, *BoxSize=None*, *rsd=None*, *select=None*)

Bases: *`nbodykit.core.datasource.DataSource`*

Class to read field data from a HDF5 FOFGroup data file

Notes

- *h5py* must be installed to use this data source.

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`MissingColumn`	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
`open`([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
`parallel_read`(columns[, full])	Override this function for complex, large data sets.
`readall`()	

`__init__`(*path*, *m0*, *dataset='FOFGroups'*, *BoxSize=None*, *rsd=None*, *select=None*)

read data from a HDF5 FOFGroup file

Parameters **path** : str

the file path to load the data from

m0 : float

the mass unit

dataset : str, optional

the name of the dataset in HDF5 file (default: FOFGroups)

BoxSize : BoxSizeParser, optional

overide the size of the box; can be a scalar or a three-vector

rsd : { 'x', 'y', 'z' }, optional

direction to do redshift distortion

select : Query, optional

row selection based on conditions specified as string

classmethod `fill_schema`()

`logger` = <logging.Logger object>

`plugin_name` = 'FOFGroups'

`readall`()

`schema` = <ConstructorSchema: 8 parameters (6 optional)>

nbodykit.core.datasource.FastPM module

class `nbodykit.core.datasource.FastPM.`**`FastPMDataSource`**(*path, BoxSize=None, bunchsize=4194304, rsd=None, lightcone=False, potentialRSD=False, velocityRSD=True*)

Bases: *`nbodykit.core.datasource.DataSource`*

DataSource to read snapshot files of the FastPM simulation

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`MissingColumn`	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	Fill the attribute schema associated with this class
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
`open`([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
`parallel_read`(columns[, full])	
`readall`()	Override to provide a method to read all available data at once

`__init__`(*path*, *BoxSize=None*, *bunchsize=4194304*, *rsd=None*, *lightcone=False*, *potentialRSD=False*, *velocityRSD=True*)

read snapshot files of the FastPM simulation

Parameters **path** : str

the file path to load the data from

BoxSize : BoxSizeParser, optional

override the size of the box; can be a scalar or a three vector

bunchsize : int, optional

number of particles to read per rank in a bunch (default: 4194304)

rsd : { 'x', 'y', 'z' }, optional

direction to do redshift distortion

lightcone : bool, optional

potential displacement for lightcone (default: False)

potentialRSD : bool, optional

potential included in file (default: False)

velocityRSD : bool, optional

velocity included in file (default: True)

classmethod `fill_schema`()

Fill the attribute schema associated with this class

`logger` = **<logging.Logger object>**

`parallel_read`(*columns*, *full=False*)

`plugin_name` = **'FastPM'**

`schema` = **<ConstructorSchema: 9 parameters (8 optional)>**

`nbodykit.core.datasource.FastPM.`**`interpMake`**(*f*, *xmin*, *xmax*, *steps*)

nbodykit.core.datasource.Gadget module

class `nbodykit.core.datasource.Gadget.`**`GadgetDataSource`**(*path*, *BoxSize*, *ptype=[]*, *posdtype='f4'*, *veldtype='f4'*, *iddtype='u8'*, *massdtype='f4'*, *rsd=None*, *bunchsize=4194304*)

Bases: *`nbodykit.core.datasource.DataSource`*

A DataSource to read a flavor of Gadget 2 files (experimental)

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`MissingColumn`	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
`open`([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
`parallel_read`(columns[, full])	read data in parallel. if Full is True, neglect bunchsize.
`readall`()	Override to provide a method to read all available data at once

`__init__`(*path*, *BoxSize*, *ptype=[]*, *posdtype='f4'*, *veldtype='f4'*, *iddtype='u8'*, *massdtype='f4'*, *rsd=None*, *bunchsize=4194304*)

read a flavor of Gadget 2 files (experimental)

Parameters **path** : str

the file path to load the data from

BoxSize : BoxSizeParser

the size of the isotropic box, or the sizes of the 3 box dimensions

ptype : int, optional

the particle types to use (default: [])

posdtype : str, optional

dtype of position (default: f4)

veldtype : str, optional

dtype of velocity (default: f4)

iddtype : str, optional

dtype of id (default: u8)

massdtype : str, optional

dtype of mass (default: f4)

rsd : { 'x', 'y', 'z' }, optional

direction to do redshift distortion

bunchsize : int, optional

number of particles to read per rank in a bunch (default: 4194304)

classmethod `fill_schema`()

`logger` = <logging.Logger object>

`parallel_read`(*columns*, *full=False*)

read data in parallel. if Full is True, neglect bunchsize.

`plugin_name` = 'Gadget'

`schema` = <ConstructorSchema: 11 parameters (9 optional)>

class `nbodykit.core.datasource.Gadget.`**`GadgetGroupTabDataSource`**(*path*, *BoxSize*, *mpch=1000.0*, *posdtype='f4'*, *veldtype='f4'*, *massdtype='f4'*, *rsd=None*, *bunchsize=4194304*)

Bases: *`nbodykit.core.datasource.DataSource`*

A DataSource to read a flavor of Gadget 2 FOF catalogs (experimental)

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`MissingColumn`	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
`open`([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
`parallel_read`(columns[, full])	Override this function for complex, large data sets.
`read`(columns[, full])	read data in parallel. if Full is True, neglect bunchsize.
`readall`()	Override to provide a method to read all available data at once

`__init__`(*path*, *BoxSize*, *mpch=1000.0*, *posdtype='f4'*, *veldtype='f4'*, *massdtype='f4'*, *rsd=None*, *bunchsize=4194304*)

read a flavor of Gadget 2 FOF catalogs (experimental)

Parameters **path** : str

the file path to load the data from

BoxSize : BoxSizeParser

the size of the isotropic box, or the sizes of the 3 box dimensions

mpch : float, optional

Mpc/h in code unit (default: 1000.0)

posdtype : optional

dtype of position (default: f4)

veldtype : optional

dtype of velocity (default: f4)

massdtype : optional

dtype of mass (default: f4)

rsd : { 'x', 'y', 'z' }, optional

direction to do redshift distortion

bunchsize : int, optional

number of particles to read per rank in a bunch (default: 4194304)

classmethod `fill_schema`()

`logger` = <logging.Logger object>

`plugin_name` = 'GadgetGroupTab'

`read`(*columns*, *full=False*)
read data in parallel. if Full is True, neglect bunchsize.

`schema` = <ConstructorSchema: 10 parameters (8 optional)>

nbodykit.core.datasource.HaloLabel module

class `nbodykit.core.datasource.HaloLabel.`**`HaloLabel`**(*path*, *bunchsize=4194304*)
Bases: `nbodykit.core.datasource.DataSource`

DataSource for reading a file of halo labels (halo id per particle), as generated the FOF algorithm

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`MissingColumn`	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
`open`([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
`parallel_read`(columns[, full])	
`readall`()	Override to provide a method to read all available data at once

`__init__`(*path*, *bunchsize=4194304*)
read a file of halo labels (halo id per particle), as generated the FOF algorithm

Parameters **path** : str

the file path to load the data from

bunchsize : int, optional

number of particle to read in a bunch (default: 4194304)

classmethod `fill_schema`()

`logger` = <logging.Logger object>

parallel_read(*columns*, *full=False*)

plugin_name = 'HaloLabel'

schema = <ConstructorSchema: 4 parameters (3 optional)>

nbodykit.core.datasource.MultiFile module

class nbodykit.core.datasource.MultiFile.**MultiFileDataSource**(*filetype*, *path*, *args={}*, *transform={}*, *enable_dask=False*)

Bases: *nbodykit.core.datasource.DataSource*

Attributes

size	The total size of the DataSource.
string	A unique identifier for the plugin, using the id()

Methods

BoxSizeParser(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
MissingColumn	
create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	
from_config(parsed)	Instantiate a plugin from this extension point,
keep_cache()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
open([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
parallel_read(columns[, full])	
readall()	Override to provide a method to read all available data at once

__init__(*filetype*, *path*, *args={}*, *transform={}*, *enable_dask=False*)

read snapshot files a multitype file

Parameters **filetype :**

the file path to load the data from

path :

the file path to load the data from

args : dict, optional

the file path to load the data from (default: {})

transform : dict, optional

transformation (default: {})

enable_dask : bool, optional

use dask (default: False)

classmethod fill_schema()

logger = <logging.Logger object>

parallel_read(*columns*, *full=False*)

plugin_name = 'MultiFile'

schema = <ConstructorSchema: 7 parameters (5 optional)>

nbodykit.core.datasource.Pandas module

class nbodykit.core.datasource.Pandas.**PandasDataSource**(*path, names, BoxSize, usecols=None, poscols=['x', 'y', 'z'], velcols=None, rsd=None, posf=1.0, velf=1.0, select=None, ftype='auto'*)

Bases: *nbodykit.core.datasource.DataSource*

Class to read data from a plaintext file using *pandas.read_csv* or a 'pandas-flavored' *HDF5* file using *pandas.read_hdf*

Reading method is guessed from the file type, or specified via the *ftype* argument.

Notes

- *pandas* must be installed to use
- when reading plaintext files, the file is assumed to be space-separated
- commented lines must begin with #, with all other lines providing data values to be read
- *names* parameter must be equal to the number of data columns, otherwise behavior is undefined

Attributes

size	The total size of the DataSource.
string	A unique identifier for the plugin, using the id()

Methods

BoxSizeParser(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
MissingColumn	
create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	Fill the attribute schema associated with this class
from_config(parsed)	Instantiate a plugin from this extension point,
keep_cache()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
open([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
parallel_read(columns[, full])	Override this function for complex, large data sets.
readall()	Read all available data, returning a dictionary

__init__(*path, names, BoxSize, usecols=None, poscols=['x', 'y', 'z'], velcols=None, rsd=None, posf=1.0, velf=1.0, select=None, ftype='auto'*)

read data from a plaintext or HDF5 file using Pandas

Parameters **path** : str

the file path to load the data from

names : str

names of columns in text file or name of the data group in hdf5 file

BoxSize : BoxSizeParser

the size of the isotropic box, or the sizes of the 3 box dimensions

usecols : str, optional

only read these columns from file

poscols : str, optional

names of the position columns (default: ['x', 'y', 'z'])

velcols : str, optional

names of the velocity columns

rsd : { 'x', 'y', 'z' }, optional

direction to do redshift distortion

posf : float, optional

factor to scale the positions (default: 1.0)

velf : float, optional

factor to scale the velocities (default: 1.0)

select : Query, optional

row selection based on conditions specified as string, i.e., "Mass > 1e14"

ftype : { 'hdf5', 'text', 'auto' }, optional

format of the Pandas storage container. auto is to guess from the file name. (default: auto)

classmethod `fill_schema`()

Fill the attribute schema associated with this class

`logger` = <logging.Logger object>

`plugin_name` = 'Pandas'

`readall`()

Read all available data, returning a dictionary

This provides `Position` and optionally `Velocity` columns, as well as any columns listed in `names`

`schema` = <ConstructorSchema: 13 parameters (10 optional)>

nbodykit.core.datasource.PlainText module

class `nbodykit.core.datasource.PlainText.`**`PlainTextDataSource`**(*path, names, BoxSize, usecols=None, poscols=['x', 'y', 'z'], velcols=None, rsd=None, posf=1.0, velf=1.0, select=None*)

Bases: *`nbodykit.core.datasource.DataSource`*

DataSource to read a plaintext file, using *numpy.recfromtxt* to do the reading

Notes

- data file is assumed to be space-separated
- commented lines must begin with #, with all other lines providing data values to be read
- *names* parameter must be equal to the number of data columns, otherwise behavior is undefined

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`MissingColumn`	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
`open`([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
`parallel_read`(columns[, full])	Override this function for complex, large data sets.
`readall`()	Read all available data, returning a dictionary

`__init__`(*path, names, BoxSize, usecols=None, poscols=['x', 'y', 'z'], velcols=None, rsd=None, posf=1.0, velf=1.0, select=None*)

read data from a plaintext file using numpy

Parameters **path** : str

the file path to load the data from

names : str

names of columns in text file or name of the data group in hdf5 file

BoxSize : BoxSizeParser

the size of the isotropic box, or the sizes of the 3 box dimensions

usecols : str, optional

only read these columns from file

poscols : str, optional

names of the position columns (default: ['x', 'y', 'z'])

velcols : str, optional

names of the velocity columns

rsd : { 'x', 'y', 'z' }, optional

direction to do redshift distortion

posf : float, optional

factor to scale the positions (default: 1.0)

velf : float, optional

factor to scale the velocities (default: 1.0)

select : Query, optional

row selection based on conditions specified as string, i.e., "Mass > 1e14"

classmethod `fill_schema`()

`logger` = <logging.Logger object>

`plugin_name` = 'PlainText'

`readall`()

Read all available data, returning a dictionary

This provides `Position` and optionally `Velocity` columns

`schema` = <ConstructorSchema: 12 parameters (9 optional)>

nbodykit.core.datasource.RaDecRedshift module

class `nbodykit.core.datasource.RaDecRedshift.`**`RaDecRedshiftDataSource`**(*path, names, unit_sphere=False, usecols=None, sky_cols=['ra', 'dec'], z_col='z', weight_col=None, degrees=False, select=None, nbar_col=None, colmap={}*)

Bases: *`nbodykit.core.datasource.DataSource`*

DataSource designed to handle reading (ra, dec, redshift) from a plaintext file, using *pandas.read_csv*

- Returns the Cartesian coordinates corresponding to (ra, dec, redshift) as the *Position* column.
- If *unit_sphere* = *True*, the Cartesian coordinates are on the unit sphere, so the the redshift information is not used

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a

Continue

Table 1.73 – continued from previous page

`MissingColumn`	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
`open`([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
`parallel_read`(columns[, full])	Override this function for complex, large data sets.
`readall`()	Read all available data, returning a dictionary

`__init__`(*path, names, unit_sphere=False, usecols=None, sky_cols=['ra', 'dec'], z_col='z', weight_col=None, degrees=False, select=None, nbar_col=None, colmap={}*)

read (ra, dec, z) from a plaintext file, returning Cartesian coordinates

Parameters **path** : str

the file path to load the data from

names : str

the names of columns in text file

unit_sphere : bool, optional

if True, return Cartesian coordinates on the unit sphere (default: False)

usecols : str, optional

only read these columns from file

sky_cols : str, optional

names of the columns specifying the sky coordinates (default: ['ra', 'dec'])

z_col : str, optional

name of the column specifying the redshift coordinate (default: z)

weight_col : str, optional

name of the column specifying the *weight* for each object

degrees : bool, optional

set this flag if the input (ra, dec) are in degrees (default: False)

select : Query, optional

row selection based on conditions specified as string

nbar_col : str, optional

name of the column specifying the *nbar* value for each object

colmap : dict, optional

dictionary that maps input columns to output columns (default: {})

classmethod `fill_schema`()

`logger` = **<logging.Logger object>**

`plugin_name` = **'RaDecRedshift'**

readall()

Read all available data, returning a dictionary

This provides the following columns:

`Ra` : right ascension (in radians) `Dec` : declination (in radians) `Redshift` : the redshift `Position` : cartesian coordinates computed from angular coords + redshift

And optionally, the *Weight* and *Nbar* columns

schema = <ConstructorSchema: 13 parameters (11 optional)>

nbodykit.core.datasource.ShiftedObserver module

class `nbodykit.core.datasource.ShiftedObserver.`**ShiftedObserverDataSource**(*datasource*, *translate*, *rsd=False*)

Bases: `nbodykit.core.datasource.DataSource`

Class to shift the observer using periodic box data and impose redshift-space distortions along the the resulting observer's line-of-sight

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`MissingColumn`	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
`open`([defaults])	Overload the *open* function to define the *DataStream* on
`parallel_read`(columns[, full])	Override this function for complex, large data sets.
`readall`()	Override to provide a method to read all available data at once
`transform`(read_func)	Decorator to transform data output of the *DataStream*

__init__(*datasource*, *translate*, *rsd=False*)

establish an explicit observer (outside the box) for a periodic box

Parameters datasource : DataSource.from_config

the data to translate in order to impose an explicit observer line-of-sight

translate : float

translate the input data by this vector

rsd : bool, optional

if *True*, impose redshift distortions along the observer's line-of-sight (default: False)

classmethod fill_schema()

logger = <logging.Logger object>

open(*defaults={}*)
Overload the *open* function to define the *DataStream* on the *datasource* attribute and to use the decorated *read* function

plugin_name = 'ShiftedObserver'

schema = <ConstructorSchema: 5 parameters (3 optional)>

transform(*read_func*)
Decorator to transform data output of the *DataStream*

Parameters read_func : callable

the original *DataStream.read* function

nbodykit.core.datasource.Subsample module

class nbodykit.core.datasource.Subsample.**SubsampleDataSource**(*path*, *dataset='Subsample'*, *BoxSize=None*, *rsd=None*)

Bases: *nbodykit.core.datasource.DataSource*

Class to read field data from a HDF5 Subsample data file

Notes

•*h5py* must be installed to use this data source.

Attributes

size	The total size of the DataSource.
string	A unique identifier for the plugin, using the id()

Methods

BoxSizeParser(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
MissingColumn	
create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	
from_config(parsed)	Instantiate a plugin from this extension point,
keep_cache()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
open([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
parallel_read(columns[, full])	Override this function for complex, large data sets.
readall()	

__init__(*path*, *dataset='Subsample'*, *BoxSize=None*, *rsd=None*)
read data from a HDF5 Subsample file

Parameters path : str

the file path to load the data from

dataset : str, optional

the name of the dataset in HDF5 file (default: Subsample)

BoxSize : BoxSizeParser, optional

overide the size of the box; can be a scalar or a three-vector

rsd : { 'x', 'y', 'z' }, optional

direction to do redshift distortion

classmethod **`fill_schema`**`()`

`logger` **= <logging.Logger object>**

`plugin_name` **= 'Subsample'**

`readall``()`

`schema` **= <ConstructorSchema: 6 parameters (5 optional)>**

nbodykit.core.datasource.TPMLabel module

class `nbodykit.core.datasource.TPMLabel.`**`TPMLabel`**(*path*, *bunchsize=4194304*)

Bases: *`nbodykit.core.datasource.DataSource`*

DataSource to read file of halo labels (halo id per particle), as generated from Martin White's TPM simulations

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`MissingColumn`	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
`open`([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
`parallel_read`(columns[, full])	read data in parallel. if Full is True, neglect bunchsize.
`readall`()	Override to provide a method to read all available data at once

`__init__`(*path*, *bunchsize=4194304*)

read file of halo labels as generated from Martin White's TPM

Parameters **path** : str

the file path to load the data from

bunchsize : int, optional

number of particle to read in a bunch (default: 4194304)

classmethod **`fill_schema`**()

`logger` **= <logging.Logger object>**

`parallel_read`(*columns*, *full=False*)
read data in parallel. if Full is True, neglect bunchsize.

`plugin_name` **= 'TPMLabel'**

`schema` **= <ConstructorSchema: 4 parameters (3 optional)>**

nbodykit.core.datasource.TPMSnapshot module

class `nbodykit.core.datasource.TPMSnapshot.`**`TPMSnapshotDataSource`**(*path*, *BoxSize*, *rsd=None*, *bunchsize=4194304*)

Bases: *`nbodykit.core.datasource.DataSource`*

DataSource to read snapshot files from Martin White's TPM simulations

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`MissingColumn`	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
`open`([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
`parallel_read`(columns[, full])	read data in parallel. if Full is True, neglect bunchsize.
`readall`()	Override to provide a method to read all available data at once

`__init__`(*path*, *BoxSize*, *rsd=None*, *bunchsize=4194304*)
read snapshot files from Martin White's TPM

Parameters **path** : str

the file path to load the data from

BoxSize : BoxSizeParser

the size of the isotropic box, or the sizes of the 3 box dimensions

rsd : { 'x', 'y', 'z' }, optional

direction to do redshift distortion

bunchsize : int, optional

number of particles to read per rank in a bunch (default: 4194304)

classmethod `fill_schema()`

`logger` **= <logging.Logger object>**

`parallel_read`(*columns*, *full=False*)
read data in parallel. if Full is True, neglect bunchsize.

This supports *Position*, *Velocity* columns

`plugin_name` **= 'TPMSnapshot'**

`schema` **= <ConstructorSchema: 6 parameters (4 optional)>**

nbodykit.core.datasource.UniformBox module

class `nbodykit.core.datasource.UniformBox.UniformBoxDataSource`(*N*, *BoxSize*, *max_speed=500.0*, *mu_logM=13.5*, *sigma_logM=0.5*, *seed=None*)

Bases: *`nbodykit.core.datasource.DataSource`*

DataSource to return the following columns:

Position : uniformly distributed in box *Velocity* : uniformly distributed between `+/- max_speed` *LogMass* : normally distributed with mean *mu_logM* and std dev *sigma_logM* *Mass* : values corresponding to 10**'Log-Mass'

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`MissingColumn`	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
`open`([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
`parallel_read`(columns[, full])	Override this function for complex, large data sets.
`readall`()	Valid columns are:

`__init__`(*N*, *BoxSize*, *max_speed=500.0*, *mu_logM=13.5*, *sigma_logM=0.5*, *seed=None*)
data particles with uniform positions and velocities

Parameters **N** : int

the total number of objects in the box

BoxSize : BoxSizeParser

the size of the isotropic box, or the sizes of the 3 box dimensions

max_speed : float, optional

use uniform velocities between [-max_speed, max_speed] (in km/s) (default: 500.0)

mu_logM : float, optional

the mean log10 mass to use when generating mass values (default: 13.5)

sigma_logM : float, optional

the standard deviation of the log10 mass to use when generating mass values (default: 0.5)

seed : int, optional

the number used to seed the random number generator

classmethod **`fill_schema`**`()`

`logger` **= <logging.Logger object>**

`plugin_name` **= 'UniformBox'**

`readall``()`

Valid columns are: *Position* : uniformly distributed in box *Velocity* : uniformly distributed between `+/-` `max_speed` *LogMass* : normally distributed with mean *mu_logM* and std dev *sigma_logM* *Mass* : values corresponding to 10**'LogMass'

`schema` **= <ConstructorSchema: 8 parameters (6 optional)>**

nbodykit.core.datasource.ZeldovichSim module

class `nbodykit.core.datasource.ZeldovichSim.`**`ZeldovichSimDataSource`**`(`*nbar*, *redshift*, *BoxSize*, *Nmesh*, *bias=2.0*, *rsd=None*, *seed=None*`)`

Bases: *`nbodykit.core.datasource.DataSource`*

DataSource to return a catalog of simulated objects, using the Zel'dovich displacement field and overdensity field generated from an input power spectrum

Note: This class requires *classylss* to be installed; see https://pypi.python.org/pypi/classylss

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser(value)`	Read the *BoxSize*, enforcing that the BoxSize must be a
`MissingColumn`	

Continue

Table 1.85 – continued from previous page

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
`open`([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
`parallel_read`(columns[, full])	Return the position of the simulated particles – 'Position' is the
`readall`()	Override to provide a method to read all available data at once

`__init__`(*nbar*, *redshift*, *BoxSize*, *Nmesh*, *bias=2.0*, *rsd=None*, *seed=None*)
simulated particles using the Zel'dovich approximation

Parameters **nbar** : float

the desired number density of the catalog in the box

redshift : float

the desired redshift of the catalog

BoxSize : BoxSizeParser

the size of the isotropic box, or the sizes of the 3 box dimensions

Nmesh : int

the number of cells per box side in the gridded mesh

bias : float, optional

the linear bias factor to apply (default: 2.0)

rsd : { 'x', 'y', 'z' }, optional

the direction to add redshift space distortions to; default is no RSD

seed : int, optional

the number used to seed the random number generator

classmethod `fill_schema`()

`logger` = <logging.Logger object>

`parallel_read`(*columns*, *full=False*)
Return the position of the simulated particles – 'Position' is the only valid column

`plugin_name` = 'ZeldovichSim'

`schema` = <ConstructorSchema: 9 parameters (5 optional)>

nbodykit.core.datasource.Zheng07HOD module

class `nbodykit.core.datasource.Zheng07HOD.`**`Zheng07HodDataSource`**(*halocat*, *redshift*, *logMmin=13.031*, *sigma_logM=0.38*, *alpha=0.76*, *logM0=13.27*, *logM1=14.08*, *rsd=None*, *seed=None*)

Bases: *`nbodykit.core.datasource.DataSource`*

A *DataSource* that uses the Hod prescription of Zheng et al. 2007 to populate an input halo catalog with galaxies, and returns the (Position, Velocity) of those galaxies

The mock population is done using *halotools* (http://halotools.readthedocs.org) The Hod model is of the commonly-used form:

- logMmin: Minimum mass required for a halo to host a central galaxy
- sigma_logM: Rate of transition from <Ncen>=0 –> <Ncen>=1
- alpha: Power law slope of the relation between halo mass and <Nsat>
- logM0: Low-mass cutoff in <Nsat>
- logM1: Characteristic halo mass where <Nsat> begins to assume a power law form

See the documentation for the *halotools* builtin Zheng07 Hod model, for further details regarding the Hod

Attributes

`size`	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`MissingColumn`	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keyw
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`keep_cache`()	A context manager that forces the DataSource cache to persist, even if there are no open Da
`log_populated_stats`()	Log statistics of the populated catalog
`open`([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
`parallel_read`(columns[, full])	Return the positions of galaxies, populated into halos using an Hod
`populate_mock`()	Populate the halo catalog with a new set of galaxies.
`readall`()	Override to provide a method to read all available data at once
`update_and_populate`(**params)	Update the Hod parameters and populate a mock

`__init__`(*halocat, redshift, logMmin=13.031, sigma_logM=0.38, alpha=0.76, logM0=13.27, logM1=14.08, rsd=None, seed=None*)

Default values for Hod values from Reid et al. 2014

populate an input halo catalog with galaxies using the Zheng et al. 2007 HOD

Parameters **halocat** : DataSource.from_config

DataSource representing the *halo* catalog

redshift : float

the redshift of the

logMmin : float, optional

minimum mass required for a halo to host a central galaxy (default: 13.031)

sigma_logM : float, optional

rate of transition from <Ncen>=0 –> <Ncen>=1 (default: 0.38)

alpha : float, optional

power law slope of the relation between halo mass and <Nsat> (default: 0.76)

logM0 : float, optional

Low-mass cutoff in <Nsat> (default: 13.27)

logM1 : float, optional

characteristic halo mass where <Nsat> begins to assume a power law form (default: 14.08)

rsd : { 'x', 'y', 'z' }, optional

the direction to do the redshift distortion

seed : int, optional

the number used to seed the random number generator

classmethod `fill_schema()`

`log_populated_stats()`
Log statistics of the populated catalog

`logger` = **<logging.Logger object>**

`parallel_read(`*columns*, *full=False*`)`
Return the positions of galaxies, populated into halos using an Hod

`plugin_name` = **'Zheng07Hod'**

`populate_mock()`
Populate the halo catalog with a new set of galaxies. Each call to this function creates a new galaxy catalog, overwriting any existing catalog

This is only done on root 0, which is the only rank to have the Hod model

`schema` = **<ConstructorSchema: 11 parameters (9 optional)>**

`update_and_populate(`***params*`)`
Update the Hod parameters and populate a mock catalog using the new parameters

This is only done on root 0, which is the only rank to have the Hod model

`nbodykit.core.datasource.Zheng07HOD.model_with_random_seed(`*model*, *seed*`)`
Update the relevant functions of the model to use the specified random seed

`nbodykit.core.datasource.Zheng07HOD.set_random_seed(`*f*, *seed*`)`
Decorator to seed the random seed in the `mc_occuputation_*` functions of the HOD model instance

nbodykit.core.painter package

class `nbodykit.core.painter.Painter(`*paintbrush*`)`
Bases: *nbodykit.plugins.PluginBase*

Mount point for plugins which refer to the painting of data, i.e., gridding a field to a mesh

Plugins of this type should provide the following attributes:

plugin_name [str] A class attribute that defines the name of the plugin in the registry

register [classmethod] A class method taking no arguments that updates the `ConstructorSchema` with the arguments needed to initialize the class

paint [method] A method that performs the painting of the field.

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`basepaint`(real, position, paintbrush[, weight])	The base function for painting that is used by default.
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs pas
`fill_schema`()	The class method responsible fill the class's schema with the relevant paramete
`from_config`(parsed)	Instantiate a plugin from this extension point,
`paint`(pm, datasource)	Paint the DataSource specified to a mesh
`shiftedpaint`(real1, real2, position, paintbrush)	paint to two real fields for interlacing

`__init__`(*paintbrush*)

`basepaint`(*real*, *position*, *paintbrush*, *weight=None*)

The base function for painting that is used by default. This handles the domain decomposition steps that are necessary to complete before painting.

Parameters **pm** : `ParticleMesh`

particle mesh object that does the painting

position : array_like

the position data

paintbrush : string

picking the paintbrush. Available ones are from documentation of pm.RealField.paint().

weight : array_like, optional

the weight value to use when painting

`logger` = **<logging.Logger object>**

`paint`(*pm*, *datasource*)

Paint the DataSource specified to a mesh

Parameters **pm** : `ParticleMesh`

particle mesh object that does the painting

datasource : DataSource

the data source object representing the field to paint onto the mesh

Returns **stats** : dict

dictionary of statistics related to painting and reading of the DataSource

`required_attributes` = **['paintbrush']**

`shiftedpaint`(*real1*, *real2*, *position*, *paintbrush*, *weight=None*, *shift=0.5*)

paint to two real fields for interlacing

Submodules

nbodykit.core.painter.DefaultPainter module

class `nbodykit.core.painter.DefaultPainter.`**`DefaultPainter`**(*weight=None*, *frho=None*, *fk=None*, *normalize=False*, *setMean=None*, *paintbrush='cic'*, *interlaced=False*)

Bases: *`nbodykit.core.painter.Painter`*

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`basepaint`(real, position, paintbrush[, weight])	The base function for painting that is used by default.
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs pas
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`paint`(pm, datasource)	Paint the `DataSource` specified by `input` onto the
`shiftedpaint`(real1, real2, position, paintbrush)	paint to two real fields for interlacing

`__init__`(*weight=None*, *frho=None*, *fk=None*, *normalize=False*, *setMean=None*, *paintbrush='cic'*, *interlaced=False*)

grid the density field of an input DataSource of objects, optionally using a weight for each object.

Parameters **weight** : optional

the column giving the weight for each object

frho : str, optional

A python expresion for transforming the real space density field. variables: rho. example: 1 + (rho - 1)**2

fk : str, optional

A python expresion for transforming the fourier space density field. variables: k, kx, ky, kz. example: exp(-(k * 0.5)**2). applied before frho

normalize : bool, optional

Normalize the field to set mean == 1. Applied before fk. (default: False)

setMean : float, optional

Set the mean. Applied after normalize.

paintbrush : str, optional

select a paint brush. Default is to defer to the choice of the algorithm that uses the painter. (default: cic)

interlaced : bool, optional

interlaced. (default: False)

classmethod `fill_schema`()

`logger` = <logging.Logger object>

`paint`(*pm*, *datasource*)

Paint the `DataSource` specified by `input` onto the `ParticleMesh` specified by `pm`

Parameters **pm** : `ParticleMesh`

particle mesh object that does the painting

datasource : `DataSource`

the data source object representing the field to paint onto the mesh

Returns **stats** : dict

dictionary of statistics, usually only containing *Ntot*

`plugin_name` = 'DefaultPainter'

`schema` = <ConstructorSchema: 8 parameters (8 optional)>

nbodykit.core.painter.MomentumPainter module

class `nbodykit.core.painter.MomentumPainter.`**`MomentumPainter`**(*velocity_component*, *moment=1*, *paintbrush='cic'*)

Bases: *`nbodykit.core.painter.Painter`*

A class to paint the mass-weighted velocity field (momentum)

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`basepaint`(real, position, paintbrush[, weight])	The base function for painting that is used by default.
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs pas
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`paint`(pm, datasource)	Paint the `DataSource` specified by `input` onto the
`shiftedpaint`(real1, real2, position, paintbrush)	paint to two real fields for interlacing

`__init__`(*velocity_component*, *moment=1*, *paintbrush='cic'*)

grid the velocity-weighted density field (momentum) field of an input DataSource of objects

Parameters **velocity_component** : { 'x', 'y', 'z' }

which velocity component to grid, either 'x', 'y', 'z'

moment : int, optional

the moment of the velocity field to paint, i.e., *moment=1* paints density*velocity, *moment=2* paints density*velocity^2 (default: 1)

paintbrush : str, optional

select a paint brush. Default is to defer to the choice of the algorithm that uses the painter. (default: cic)

classmethod **`fill_schema`**()

`logger` **= <logging.Logger object>**

`paint`(*pm*, *datasource*)

Paint the `DataSource` specified by `input` onto the `ParticleMesh` specified by `pm`

Parameters **pm** : `ParticleMesh`

particle mesh object that does the painting

datasource : `DataSource`

the data source object representing the field to paint onto the mesh

Returns **stats** : dict

dictionary of statistics, usually only containing *Ntot*

`plugin_name` **= 'MomentumPainter'**

`schema` **= <ConstructorSchema: 4 parameters (3 optional)>**

nbodykit.core.source package

class `nbodykit.core.source.`**`Painter`**(*frho=None*, *fk=None*, *normalize=False*, *setMean=None*, *paintbrush='cic'*, *interlaced=False*)

Bases: `object`

Painter object to help Sources convert results from Source.read to a RealField

Methods

`from_config`(d)	Initialize the class from a dictionary of parameters
`paint`(source, pm)	Paint the input *source* to the mesh specified by *pm*
`transform`(source, real)	Apply (in-place) transformations to the real-space field

`__init__`(*frho=None*, *fk=None*, *normalize=False*, *setMean=None*, *paintbrush='cic'*, *interlaced=False*)

classmethod **`from_config`**(*d*)

Initialize the class from a dictionary of parameters

`paint`(*source*, *pm*)

Paint the input *source* to the mesh specified by *pm*

Parameters **source** : *Source* or a subclass

the source object from which the default

pm : pmesh.pm.ParticleMesh

the particle mesh object

Returns **real** : pmesh.pm.RealField

the painted real field

transform(*source*, *real*)
Apply (in-place) transformations to the real-space field specified by *real*

class nbodykit.core.source.**Source**(**args*, ***kwargs*)
Bases: *nbodykit.plugins.PluginBase*

A base class to represent an object that combines the processes of reading / generating data and painting to a RealField

Attributes

BoxSize	A 3-vector specifying the size of the box for this source
attrs	
columns	
string	A unique identifier for the plugin, using the id()

Methods

compute(*args, **kwargs)	Our version of `dask.compute()` that computes
create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	The class method responsible fill the class's schema with the relevant parameters from the _
from_config(parsed)	Instantiate a plugin from this extension point,
paint(pm)	
read(columns)	

BoxSize
A 3-vector specifying the size of the box for this source

attrs

columns

static compute(**args*, ***kwargs*)
Our version of `dask.compute()` that computes multiple delayed dask collections at once

Parameters args : object

Any number of objects. If the object is a dask collection, it's computed and the result is returned. Otherwise it's passed through unchanged.

Notes

The dask default optimizer induces too many (unnecesarry) IO calls – we turn this off feature off by default.

Eventually we want our own optimizer probably.

logger = <logging.Logger object>

paint(*pm*)

read(*columns*)

Submodules

nbodykit.core.source.Grid module

class `nbodykit.core.source.Grid.`**`GridSource`**(*path*, *dataset*, *attrs={}*, *painter=<nbodykit.core.source.Painter object>*)

Bases: *`nbodykit.core.source.Source`*

Attributes

`BoxSize`	A 3-vector specifying the size of the box for this source
`attrs`	
`columns`	
`string`	A unique identifier for the plugin, using the `id()`

Methods

`compute`(*args, **kwargs)	Our version of `dask.compute()` that computes
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`paint`(pm)	
`read`(columns)	

`__init__`(*path*, *dataset*, *attrs={}*, *painter=<nbodykit.core.source.Painter object>*)

read snapshot files a multitype file

Parameters path :

the file path to load the data from

dataset :

dataset

attrs : dict, optional

override attributes from the file (default: {})

painter : Painter.from_config, optional

painter parameters (default: <nbodykit.core.source.Painter object at 0x7fde5bf85320>) The 6 subfields are:

paintbrush [{ 'cubic', 'LANCZOS3', 'db12', 'sym20', 'sym6', 'db6', 'LANCZOS2', 'QUADRATIC', 'TSC', 'DB20', 'db20', 'LINEAR', 'CUBIC', 'lanczos3', 'sym12', 'SYM6', 'CIC', 'lanczos2', 'SYM12', 'DB6', 'tsc', 'cic', 'linear', 'DB12', 'SYM20', 'quadratic' }] paintbrush

frho [str] A python expresion for transforming the real space density field. variables: rho. example: 1 + (rho - 1)**2

fk [str] A python expresion for transforming the fourier space density field. variables: k, kx, ky, kz. example: exp(-(k * 0.5)**2). applied before frho

normalize [bool] Normalize the field to set mean == 1. Applied before fk.

setMean [float] Set the mean. Applied after normalize.

interlaced [bool] interlaced.

attrs

columns

classmethod fill_schema()

logger = **<logging.Logger object>**

paint(*pm*)

plugin_name = **'Source.Grid'**

read(*columns*)

schema = **<ConstructorSchema: 6 parameters (4 optional)>**

nbodykit.core.source.Particle module

class nbodykit.core.source.Particle.**ParticleSource**(*filetype*, *path*, *args={}*, *transform={}*, *attrs={}*, *painter=<nbodykit.core.source.Painter object>*)

Bases: *nbodykit.core.source.Source*

Attributes

BoxSize	A 3-vector specifying the size of the box for this source
attrs	
columns	
string	A unique identifier for the plugin, using the id()

Methods

compute(*args, **kwargs)	Our version of dask.compute() that computes
create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	
from_config(parsed)	Instantiate a plugin from this extension point,
paint(pm)	
read(columns)	

__init__(*filetype*, *path*, *args={}*, *transform={}*, *attrs={}*, *painter=<nbodykit.core.source.Painter object>*)

read snapshot files a multitype file

Parameters **filetype :**

the file path to load the data from

path :

the file path to load the data from

args : dict, optional

the file path to load the data from (default: {})

transform : dict, optional

data transformation (default: {})

attrs : dict, optional

override attributes from the file (default: {})

painter : Painter.from_config, optional

painter parameters (default: <nbodykit.core.source.Painter object at 0x7fde5bf852b0>)

`attrs`

`columns`

classmethod `fill_schema`()

`logger` = **<logging.Logger object>**

`paint`(*pm*)

`plugin_name` = **'Source.Particle'**

`read`(*columns*)

`schema` = **<ConstructorSchema: 8 parameters (6 optional)>**

nbodykit.core.transfer package

class `nbodykit.core.transfer.`**`Transfer`**(**args*, ***kwargs*)

Bases: *`nbodykit.plugins.PluginBase`*

Mount point for plugins which apply a k-space transfer function to the Fourier transfrom of a datasource field

Plugins of this type should provide the following attributes:

plugin_name [str] class attribute that defines the name of the plugin in the registry

register [classmethod] a class method taking no arguments that updates the `ConstructorSchema` with the arguments needed to initialize the class

__call__ [method] function that will apply the transfer function to the complex array

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`__call__`(pm, complex)	Apply the transfer function to the complex field
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	The class method responsible fill the class's schema with the relevant parameters from the _
`from_config`(parsed)	Instantiate a plugin from this extension point,

`logger` = **<logging.Logger object>**

Submodules

nbodykit.core.transfer.TransferFunction module

class nbodykit.core.transfer.TransferFunction.**AnisotropicCIC**

Bases: *nbodykit.core.transfer.Transfer*

Divide by a kernel in Fourier space to account for the convolution of the gridded quantity with the CIC window function in configuration space

Attributes

string	A unique identifier for the plugin, using the id()

Methods

__call__(pm, complex)	
create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	
from_config(parsed)	Instantiate a plugin from this extension point,

__init__()

divide by a Fourier-space kernel to account for the CIC gridding window function; see Jing et al 2005 (arxiv:0409240)

classmethod fill_schema()

logger = <logging.Logger object>

plugin_name = 'AnisotropicCIC'

schema = <ConstructorSchema: 1 parameters (1 optional)>

class nbodykit.core.transfer.TransferFunction.**AnisotropicTSC**

Bases: *nbodykit.core.transfer.Transfer*

Divide by a kernel in Fourier space to account for the convolution of the gridded quantity with the TSC window function in configuration space

Attributes

string	A unique identifier for the plugin, using the id()

Methods

__call__(pm, complex)	
create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	
from_config(parsed)	Instantiate a plugin from this extension point,

`__init__``()`
divide by a Fourier-space kernel to account for the TSC gridding window function; see Jing et al 2005 (arxiv:0409240)

classmethod `fill_schema``()`

`logger` **= <logging.Logger object>**

`plugin_name` **= 'AnisotropicTSC'**

`schema` **= <ConstructorSchema: 1 parameters (1 optional)>**

class `nbodykit.core.transfer.TransferFunction.`**`CICWindow`**
Bases: *`nbodykit.core.transfer.Transfer`*

Divide by a kernel in Fourier space to account for the convolution of the gridded quantity with the CIC window function in configuration space

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`__call__`(pm, complex)	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,

`__init__``()`
divide by a Fourier-space kernel to account for the CIC gridding window function; see Jing et al 2005 (arxiv:0409240)

classmethod `fill_schema``()`

`logger` **= <logging.Logger object>**

`plugin_name` **= 'CICWindow'**

`schema` **= <ConstructorSchema: 1 parameters (1 optional)>**

class `nbodykit.core.transfer.TransferFunction.`**`NormalizeDC`**
Bases: *`nbodykit.core.transfer.Transfer`*

Normalize by the DC amplitude in Fourier space, which effectively divides by the mean in configuration space

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`__call__`(pm, complex)	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,

`__init__`()
normalize the DC amplitude in Fourier space, which effectively divides by the mean in configuration space

classmethod `fill_schema`()

`logger` = <logging.Logger object>

`plugin_name` = 'NormalizeDC'

`schema` = <ConstructorSchema: 1 parameters (1 optional)>

class `nbodykit.core.transfer.TransferFunction.`**`RemoveDC`**
Bases: *`nbodykit.core.transfer.Transfer`*

Remove the DC amplitude, which sets the mean of the field in configuration space to zero

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`__call__`(pm, complex)	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,

`__init__`()
remove the DC amplitude in Fourier space, which sets the mean of the field in configuration space to zero

classmethod `fill_schema`()

`logger` = <logging.Logger object>

`plugin_name` = 'RemoveDC'

`schema` = <ConstructorSchema: 1 parameters (1 optional)>

class `nbodykit.core.transfer.TransferFunction.`**`TSCWindow`**
Bases: *`nbodykit.core.transfer.Transfer`*

Divide by a kernel in Fourier space to account for the convolution of the gridded quantity with the CIC window function in configuration space

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`__call__`(pm, complex)	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,

`__init__`()
divide by a Fourier-space kernel to account for the TSC gridding window function; see Jing et al 2005 (arxiv:0409240)

classmethod `fill_schema`()

`logger` = <logging.Logger object>

`plugin_name` = 'TSCWindow'

`schema` = <ConstructorSchema: 1 parameters (1 optional)>

API reference

nbodykit package

class `nbodykit.`**`GlobalComm`**
Bases: `object`

The global MPI communicator

Methods

`get`()	Get the communicator, return `MPI.COMM_WORLD`
`set`(comm)	Set the communicator to the input value

classmethod `get`()
Get the communicator, return `MPI.COMM_WORLD` if the comm has not be explicitly set yet

classmethod `set`(*comm*)
Set the communicator to the input value

class `nbodykit.`**`GlobalCosmology`**
Bases: `object`

The global *`Cosmology`* instance

Methods

get()	Get the communicator, return `MPI.COMM_WORLD`
set(cosmo)	Set the communicator to the input value

classmethod get ()
Get the communicator, return `MPI.COMM_WORLD` if the comm has not be explicitly set yet

classmethod set (*cosmo*)
Set the communicator to the input value

Subpackages

nbodykit.core package

class `nbodykit.core.`**`Algorithm`**(**args*, ***kwargs*)
Bases: *`nbodykit.plugins.PluginBase`*

Mount point for plugins which provide an interface for running one of the high-level algorithms, i.e, power spectrum calculation or FOF halo finder

Plugins of this type should provide the following attributes:

plugin_name [str] A class attribute that defines the name of the plugin in the registry

register [classmethod] A class method taking no arguments that updates the `ConstructorSchema` with the arguments needed to initialize the class

run [method] function that will run the algorithm

save [method] save the result of the algorithm computed by *`Algorithm.run()`*

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	The class method responsible fill the class's schema with the relevant parameters from the _
`from_config`(parsed)	Instantiate a plugin from this extension point,
run()	Run the algorithm
save(output, result)	Save the results of the algorithm run

`logger` = **<logging.Logger object>**

`run`()
Run the algorithm

Returns **result** : tuple

the tuple of results that will be passed to *`Algorithm.save()`*

`save`(*output*, *result*)
Save the results of the algorithm run

Parameters **output** : str

the name of the output file to save results too

result : tuple

the tuple of results returned by `Algorithm.run()`

class `nbodykit.core.`**`Source`**(**args*, ***kwargs*)

Bases: `nbodykit.plugins.PluginBase`

A base class to represent an object that combines the processes of reading / generating data and painting to a RealField

Attributes

`BoxSize`	A 3-vector specifying the size of the box for this source
`attrs`	
`columns`	
`string`	A unique identifier for the plugin, using the `id()`

Methods

`compute`(*args, **kwargs)	Our version of `dask.compute()` that computes
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	The class method responsible fill the class's schema with the relevant parameters from the _
`from_config`(parsed)	Instantiate a plugin from this extension point,
`paint`(pm)	
`read`(columns)	

`BoxSize`

A 3-vector specifying the size of the box for this source

`attrs`

`columns`

static **`compute`**(**args*, ***kwargs*)

Our version of `dask.compute()` that computes multiple delayed dask collections at once

Parameters **args** : object

Any number of objects. If the object is a dask collection, it's computed and the result is returned. Otherwise it's passed through unchanged.

Notes

The dask default optimizer induces too many (unnecesarry) IO calls – we turn this off feature off by default.

Eventually we want our own optimizer probably.

`logger` **= <logging.Logger object>**

`paint`(*pm*)

read(*columns*)

class nbodykit.core.**DataSource**(**args*, ***kwargs*)

Bases: *nbodykit.core.datasource.DataSourceBase*

Mount point for plugins which refer to the reading of input files. The *read* operation occurs on a `DataStream` object, which is returned by *open()*.

Default values for any columns to read can be supplied as a dictionary argument to *open()*.

Plugins of this type should provide the following attributes:

plugin_name [str] A class attribute that defines the name of the plugin in the registry

register [classmethod] A class method taking no arguments that updates the `ConstructorSchema` with the arguments needed to initialize the class

readall: method A method to read all available data at once (uncollectively) and cache the data in memory for repeated calls to *read*

parallel_read: method A method to read data for complex, large data sets. The read operation shall be collective, with each yield generating different sections of the data source on different ranks. No caching of data takes places.

Notes

- a *Cosmology* instance can be passed to any DataSource class via the *cosmo* keyword
- the data will be cached in memory if returned via `readall()`
- the default cache behavior is for the cache to persist while an open DataStream remains, but the cache can be forced to persist via the *DataSource.keep_cache()* context manager

Attributes

size	The total size of the DataSource.
`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
MissingColumn	
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	The class method responsible fill the class's schema with the relevant parameters from the _
`from_config`(parsed)	Instantiate a plugin from this extension point,
keep_cache()	A context manager that forces the DataSource cache to persist, even if there are no open Dat
open([defaults])	Open the DataSource by returning a DataStream from which the data can be read.
parallel_read(columns[, full])	Override this function for complex, large data sets.
readall()	Override to provide a method to read all available data at once

exception MissingColumn

Bases: `Exception`

`DataSource.`**keep_cache**()

A context manager that forces the DataSource cache to persist, even if there are no open DataStream objects. This will prevent unwanted and unnecessary re-readings of the DataSource.

The below example details the intended usage. In this example, the data is cached only once, and no re-reading of the data occurs when the second stream is opened.

```
with datasource.keep_cache():
    with datasource.open() as stream1:
        [[pos]] = stream1.read(['Position'], full=True)

    with datasource.open() as stream2:
        [[vel]] = stream2.read(['Velocity'], full=True)
```

`DataSource.`**`logger`** **= <logging.Logger object>**

`DataSource.`**`open`**`(`*defaults={}*`)`

Open the DataSource by returning a DataStream from which the data can be read.

This function also specifies the default values for any columns that are not supported by the DataSource. The defaults are unique to each DataStream, but a DataSource can be opened multiple times (returning different streams) with different default values

Parameters **defaults** : dict, optional

a dictionary providing default values for a given column

Returns **stream** : DataStream

the stream object from which the data can be read via `read()` function

`DataSource.`**`parallel_read`**`(`*columns*, *full=False*`)`

Override this function for complex, large data sets. The read operation shall be collective, each yield generates different sections of the datasource. No caching of data takes places.

If the DataSource does not provide a column in *columns*, *None* should be returned.

Parameters **columns** : list of str

the list of data columns to return

full : bool, optional

if *True*, any *bunchsize* parameters will be ignored, so that each rank will read all of its specified data section at once

Returns **data** : list

a list of the data for each column in columns; if the data source does not provide a given column, that element should be *None*

Notes

- This function will be called if `DataStream.readall()` is not implemented
- The intention is for this function to handle complex and large data sets, where parallel I/O across ranks is required to avoid memory and I/O issues

`DataSource.`**`readall`**`()`

Override to provide a method to read all available data at once (uncollectively) and cache the data in memory for repeated calls to *read*

Returns **data** : dict

a dictionary of all supported data for the data source; keys give the column names and values are numpy arrays

Notes

- By default, `DataStream.read()` calls this function on the root rank to read all available data, and then scatters the data evenly across all available ranks
- The intention is to reduce the complexity of implementing a simple and small data source, for which reading all data at once is feasible

`DataSource.`**`size`**
The total size of the DataSource.

The user can set this explicitly (only once per datasource) if the size is known before `DataStream.read()` is called

class `nbodykit.core.`**`GridSource`**(**args*, ***kwargs*)
Bases: *`nbodykit.core.datasource.DataSourceBase`*

A DataSource reading directly already on a grid

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`BoxSizeParser`(value)	Read the *BoxSize*, enforcing that the BoxSize must be a
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	The class method responsible fill the class's schema with the relevant parameters from the _
`from_config`(parsed)	Instantiate a plugin from this extension point,
`read`(real)	Read into a real field

`logger` = **<logging.Logger object>**

`read`(*real*)
Read into a real field

class `nbodykit.core.`**`Painter`**(*paintbrush*)
Bases: *`nbodykit.plugins.PluginBase`*

Mount point for plugins which refer to the painting of data, i.e., gridding a field to a mesh

Plugins of this type should provide the following attributes:

plugin_name [str] A class attribute that defines the name of the plugin in the registry

register [classmethod] A class method taking no arguments that updates the `ConstructorSchema` with the arguments needed to initialize the class

paint [method] A method that performs the painting of the field.

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`basepaint`(real, position, paintbrush[, weight])	The base function for painting that is used by default.
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs pas
`fill_schema`()	The class method responsible fill the class's schema with the relevant paramete
`from_config`(parsed)	Instantiate a plugin from this extension point,
`paint`(pm, datasource)	Paint the DataSource specified to a mesh
`shiftedpaint`(real1, real2, position, paintbrush)	paint to two real fields for interlacing

`__init__`(*paintbrush*)

`basepaint`(*real*, *position*, *paintbrush*, *weight=None*)

The base function for painting that is used by default. This handles the domain decomposition steps that are necessary to complete before painting.

Parameters **pm** : `ParticleMesh`

particle mesh object that does the painting

position : array_like

the position data

paintbrush : string

picking the paintbrush. Available ones are from documentation of pm.RealField.paint().

weight : array_like, optional

the weight value to use when painting

`logger` = **<logging.Logger object>**

`paint`(*pm*, *datasource*)

Paint the DataSource specified to a mesh

Parameters **pm** : `ParticleMesh`

particle mesh object that does the painting

datasource : DataSource

the data source object representing the field to paint onto the mesh

Returns **stats** : dict

dictionary of statistics related to painting and reading of the DataSource

`required_attributes` = **['paintbrush']**

`shiftedpaint`(*real1*, *real2*, *position*, *paintbrush*, *weight=None*, *shift=0.5*)

paint to two real fields for interlacing

class `nbodykit.core.`**`Transfer`**(**args*, ***kwargs*)

Bases: *`nbodykit.plugins.PluginBase`*

Mount point for plugins which apply a k-space transfer function to the Fourier transfrom of a datasource field

Plugins of this type should provide the following attributes:

plugin_name [str] class attribute that defines the name of the plugin in the registry

register [classmethod] a class method taking no arguments that updates the `ConstructorSchema` with the arguments needed to initialize the class

__call__ [method] function that will apply the transfer function to the complex array

Attributes

`string`	A unique identifier for the plugin, using the `id()`

Methods

`__call__`(pm, complex)	Apply the transfer function to the complex field
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	The class method responsible fill the class's schema with the relevant parameters from the _
`from_config`(parsed)	Instantiate a plugin from this extension point,

`logger` **= <logging.Logger object>**

Subpackages

nbodykit.io package

class `nbodykit.io.`**`FileType`**(**args*, ***kwargs*)

Bases: *`nbodykit.plugins.PluginBase`*

Abstract base class representing a file object

Attributes

`columns`	Returns the names of the columns in the file; this defaults
`ncol`	The number of data columns in the file
`shape`	The shape of the file, which defaults to (*size*,)
`string`	A unique identifier for the plugin, using the `id()`

Methods

`asarray`()	Return a view of the file, where the fields of the
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	The class method responsible fill the class's schema with the relevant parameters from the _
`from_config`(parsed)	Instantiate a plugin from this extension point,
`get_dask`(column[, blocksize])	Return the specified column as a dask array, which

Co

Table 1.130 – continued from previous page

`keys`()	Aliased function to return `columns`
`read`(columns, start, stop[, step])	Read the specified column(s) over the given range,

asarray()

Return a view of the file, where the fields of the structured array are stacked in columns of a single numpy array

Returns FileType :

a file object that will return a numpy array with the columns representing the fields

Examples

# original file has three named fields >> ff.dtype dtype([('ra', '<f4'), ('dec', '<f4'), ('z', '<f4')]) >> ff.shape (1000,) >> ff.columns ['ra', 'dec', 'z'] >> ff[:3] array([(235.63442993164062, 59.39099884033203, 0.6225500106811523),

(140.36181640625, -1.162310004234314, 0.5026500225067139), (129.96627807617188, 45.970130920410156, 0.4990200102329254)],

dtype=(numpy.record, [('ra', '<f4'), ('dec', '<f4'), ('z', '<f4')]))

# select subset of columns and switch the ordering # and convert output to a single numpy array >> x = ff[['dec', 'ra']].asarray() >> x.dtype dtype('float32') >> x.shape (1000, 2) >> x.columns ['dec', 'ra'] >> x[:3] array([[59.39099884, 235.63442993],

[-1.16231 , 140.36181641], [45.97013092, 129.96627808]], dtype=float32)

# select only the first column (dec) >> dec = x[:,0] >> dec[:3] array([59.39099884, -1.16231 , 45.97013092], dtype=float32)

columns

Returns the names of the columns in the file; this defaults to the named fields in the file's `dtype` attribute

This will differ from the data type's named fields if a view of the file has been returned with `asarray()`

get_dask(*column*, *blocksize=100000*)

Return the specified column as a dask array, which delays the explicit reading of the data until `dask.compute()` is called

The dask array is chunked into blocks of size *blocksize*

Parameters **column** : str

the name of the column to return

blocksize : int, optional

the size of the chunks in the dask array

Returns dask.array :

the dask array holding the column, which computes the necessary functions to read the data, but delays evaluating until the user specifies

keys()

Aliased function to return `columns`

logger = <logging.Logger object>

ncol
The number of data columns in the file

read(*columns*, *start*, *stop*, *step=1*)
Read the specified column(s) over the given range, returning a structured numpy array

Parameters **columns** : str, list of str

the name of the column(s) to return

start : int

the row integer to start reading at

stop : int

the row integer to stop reading at

step : int, optional

the step size to use when reading; default is 1

Returns **data** : array_like

a numpy structured array holding the requested data

required_attributes = ['size', 'dtype']

shape
The shape of the file, which defaults to (*size*,)

Multiple dimensions can be introduced into the shape if a view of the file has been returned with *asarray()*

nbodykit.io.**io_extension_points**()
Return a dictionary of the extension points for io

This returns only the *FileType* class

Submodules

nbodykit.io.bigfile module

class nbodykit.io.bigfile.**BigFile**(*path, exclude=['header'], header='.', root='./'*)
Bases: *nbodykit.io.FileType*

A file object to handle the reading of columns of data from a bigfile file. bigfile is the default format of FastPM and MP-Gadget.

https://github.com/rainwoodman/bigfile

Attributes

columns	Returns the names of the columns in the file; this defaults
ncol	The number of data columns in the file
shape	The shape of the file, which defaults to (*size*,)
string	A unique identifier for the plugin, using the id()

Methods

asarray()	Return a view of the file, where the fields of the
create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	
from_config(parsed)	Instantiate a plugin from this extension point,
get_dask(column[, blocksize])	Return the specified column as a dask array, which
keys()	Aliased function to return columns
read(columns, start, stop[, step])	Read the specified column(s) over the given range,

__init__(*path, exclude=['header'], header='.', root='./'*)

A class to read columns of data stored in the *bigfile* format

Parameters **path** : str

the name of the file to load

exclude : str, optional

columns to exclude (default: ['header'])

header : str, optional

block to look for the meta data attributes (default: .)

root : str, optional

block to look for the meta data attributes (default: ./)

classmethod fill_schema()

logger = <logging.Logger object>

plugin_name = 'FileType.BigFile'

read(*columns*, *start*, *stop*, *step=1*)

Read the specified column(s) over the given range, as a dictionary

'start' and 'stop' should be between 0 and size, which is the total size of the binary file (in particles)

schema = <ConstructorSchema: 5 parameters (4 optional)>

nbodykit.io.binary module

class nbodykit.io.binary.**BinaryFile**(*path*, *dtype*, *offsets=None*, *header_size=0*, *size=None*)

Bases: *nbodykit.io.FileType*

A file object to handle the reading of columns of data from a binary file.

Warning: This assumes the data is stored in a column-major format

Attributes

columns	Returns the names of the columns in the file; this defaults
ncol	The number of data columns in the file
shape	The shape of the file, which defaults to (*size*,)
string	A unique identifier for the plugin, using the id()

Methods

`asarray`()	Return a view of the file, where the fields of the
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`get_dask`(column[, blocksize])	Return the specified column as a dask array, which
`keys`()	Aliased function to return `columns`
`read`(columns, start, stop[, step])	Read the specified column(s) over the given range,

`__init__`(*path*, *dtype*, *offsets=None*, *header_size=0*, *size=None*)
a binary file reader

Parameters **path** : str

the name of the binary file to load

dtype : tuple

list of tuples of (name, dtype) to be converted to a numpy.dtype

offsets : dict, optional

a dictionary giving the byte offsets for each column in the file

header_size : int, optional

the size of the header of the in bytes (default: 0)

size : optional

an int giving the file size or a function that takes a single argument, the name of the file, and returns the size

classmethod `fill_schema`()

`logger` = <logging.Logger object>

`plugin_name` = 'BinaryFile'

`read`(*columns*, *start*, *stop*, *step=1*)
Read the specified column(s) over the given range, as a dictionary

'start' and 'stop' should be between 0 and `size`, which is the total size of the binary file (in particles)

`schema` = <ConstructorSchema: 6 parameters (4 optional)>

`nbodykit.io.binary.`**`getsize`**(*filename*, *header_size*, *rowsize*)
The default method to determine the size of the binary file

The "size" is defined as the number of rows, where each row has of size of *rowsize* in bytes.

Parameters **filename** : str

the name of the binary file

header_size : int

the size of the header in bytes, which will be skipped when determining the number of rows

rowsize : int

the size of the data in each row in bytes

Raises ValueError :

If the function determines a fractional number of rows

Notes

- This assumes the input file is not compressed
- This function does not depend on the layout of the binary file, i.e., if the data is formatted in actual rows or not

nbodykit.io.csv module

class `nbodykit.io.csv.`**`CSVFile`**(*path*, *names*, *blocksize=33554432*, *dtype={}*, *delim_whitespace=True*, *header=None*, ***config*)

Bases: *`nbodykit.io.FileType`*

A file object to handle the reading of columns of data from a CSV file

Internally, this class uses `dask.dataframe.read_csv()` (which uses func:*pandas.read_csv*) to partition the CSV file into chunks, and data is only read from the relevant chunks of the file.

This setup provides a significant speed-up when reading from the end of the file, since the entirety of the data does not need to be read first.

The class supports any of the configuration keywords that can be passed to `pandas.read_csv()`

Warning: This assumes the delimiter for separate lines is the newline character.

Attributes

`columns`	Returns the names of the columns in the file; this defaults
`ncol`	The number of data columns in the file
`shape`	The shape of the file, which defaults to (*size*,)
`string`	A unique identifier for the plugin, using the `id()`

Methods

`asarray`()	Return a view of the file, where the fields of the
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`get_dask`(column[, blocksize])	Return the specified column as a dask array, which
`keys`()	Aliased function to return `columns`
`read`(columns, start, stop[, step])	Read the specified column(s) over the given range,

`__init__`(*path*, *names*, *blocksize=33554432*, *dtype={}*, *delim_whitespace=True*, *header=None*, ***config*)

a csv file reader

Parameters path : str

the name of the file to load

names : str

the names of each column in the csv file

blocksize : int, optional

internally partition the CSV file into blocks of this size (in bytes) (default: 33554432)

dtype : optional

a dictionary providing data types for the various columns; data types not provided are inferred from the file (default: {})

delim_whitespace : bool, optional

set to True if the input file is space-separated (default: True)

header : optional

the type of header in the CSV file; if no header, set to None

classmethod `fill_schema()`

`logger` **= <logging.Logger object>**

`plugin_name` **= 'CSVFile'**

`read`(*columns*, *start*, *stop*, *step=1*)

Read the specified column(s) over the given range, as a dictionary

'start' and 'stop' should be between 0 and `size`, which is the total size of the file (in particles)

`schema` **= <ConstructorSchema: 7 parameters (5 optional)>**

nbodykit.io.stack module

class `nbodykit.io.stack.FileStack`(*path*, *filetype*, ***kwargs*)

Bases: `nbodykit.io.FileType`

A class that offers a continuous view of a stack of subclasses of `FileType` instances

Attributes

`attrs`	
`columns`	Returns the names of the columns in the file; this defaults
`ncol`	The number of data columns in the file
`nfiles`	The number of files in the FileStack
`shape`	The shape of the file, which defaults to (*size*,)
`string`	A unique identifier for the plugin, using the `id()`

Methods

`asarray()`	Return a view of the file, where the fields of the
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema()`	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`from_files`(files[, sizes])	Create a FileStack from an existing list of files

Continued on next pa

Table 1.138 – continued from previous page

get_dask(column[, blocksize])	Return the specified column as a dask array, which
keys()	Aliased function to return columns
read(columns, start, stop[, step])	Read the specified column(s) over the given range,

__init__(*path*, *filetype*, ***kwargs*)
a continuous view of a stack of files

Parameters **path** : str

a list of files or a glob-like pattern

filetype :

the file type class

attrs

classmethod fill_schema()

classmethod from_files(*files*, *sizes=[]*)
Create a FileStack from an existing list of files and optionally a list of the sizes of those files

logger = <logging.Logger object>

nfiles
The number of files in the FileStack

plugin_name = 'FileStack'

read(*columns*, *start*, *stop*, *step=1*)
Read the specified column(s) over the given range, as a dictionary

'start' and 'stop' should be between 0 and size, which is the total size of the file (in particles)

schema = <ConstructorSchema: 3 parameters (1 optional)>

nbodykit.io.tools module

nbodykit.io.tools.**csv_partition_sizes**(*filename*, *blocksize*, *delimiter='\n'*)
From a filename and preferred blocksize in bytes, return the number of rows in each partition

This divides the input file into partitions with size roughly equal to blocksize, reads the bytes, and counts the number of delimiters

Parameters **filename** : str

the name of the CSV file to load

blocksize : int

the desired number of bytes per block

delimiter : str, optional

the character separating lines; default is the newline character

Returns **nrows** : list of int

the list of the number of rows in each block

nbodykit.io.tools.**get_file_slice**(*sizes*, *start*, *stop*)
Return the list of file numbers that must be accessed to return data between *start* and *slice*, where these indices are defined in terms of the global catalog indexing

Parameters **sizes** : array_like

the sizes of each file in the file stack

start : int

the global index to begin the slice

stop : int

the global index to stop the slice

Returns **fnums** : list

the list of integers specifying the relevant file numbers that must be accessed

`nbodykit.io.tools.`**`get_slice_size`**(*start*, *stop*, *step*)

Utility function to return the size of an array slice

Parameters **start** : int

the beginning of the slice

stop : int

the end of the slice

step : int

the slice step size

Returns **N** : int

the total size of the slice

`nbodykit.io.tools.`**`global_to_local_slice`**(*sizes*, *start*, *stop*, *fnum*)

Convert a global slice, specified by *start* and *stop* to the corresponding local indices of the file specified by *fnum*

Parameters **sizes** : array_like

the sizes of each file in the file stack

start : int

the global index to begin the slice

stop : int

the global index to stop the slice

fnum : int

the file number that defines the desired local indexing

Returns **local_start, local_stop** : int

the local start and stop slice values

`nbodykit.io.tools.`**`infer_csv_dtype`**(*path*, *names*, *nrows=10*, ***config*)

Read the first few lines of the specified CSV file to determine the data type

Parameters **path** : str

the name of the CSV file to load

names : list of str

the list of the names of the columns in the CSV file

nrows : int, optional

the number of rows to read from the file in order to infer the data type; default is 10

**config : key, value pairs

additional keywords to pass to `pandas.read_csv()`

Returns dtype : dict

dictionary holding the dtype for each name in *names*

nbodykit.io.tpm module

class `nbodykit.io.tpm.`**`TPMBinaryFile`**(*path*, *precision='f4'*)

Bases: *`nbodykit.io.binary.BinaryFile`*

Read snapshot files from Martin White's TPM simulations, which are stored in a binary format

Attributes

`columns`	Returns the names of the columns in the file; this defaults
`ncol`	The number of data columns in the file
`shape`	The shape of the file, which defaults to (*size*,)
`string`	A unique identifier for the plugin, using the `id()`

Methods

`asarray`()	Return a view of the file, where the fields of the
`create`(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
`fill_schema`()	
`from_config`(parsed)	Instantiate a plugin from this extension point,
`get_dask`(column[, blocksize])	Return the specified column as a dask array, which
`keys`()	Aliased function to return `columns`
`read`(columns, start, stop[, step])	Read the specified column(s) over the given range,

`__init__`(*path*, *precision='f4'*)

read binary snapshot files from Martin White's TPM snapshots

Parameters path : str

the name of the file to load

precision : { 'f8', 'f4' }, optional

precision of floating point numbers (default: f4)

classmethod `fill_schema`()

`logger` = **<logging.Logger object>**

`plugin_name` = **'FileType.TPM'**

`schema` = **<ConstructorSchema: 3 parameters (2 optional)>**

nbodykit.plugins package

`nbodykit.plugins.`**`ListPluginsAction`**(*extension_type*, *comm*)

Return a `argparse.Action` that prints the help message for the class specified by *extension_type*

This action can take any number of arguments. If no arguments are provided, it prints the help for all registered plugins of type *extension_type*

`nbodykit.plugins.`**`MetaclassWithHooks`**(*meta*, **hooks*)

Function to return a subclass of the metaclass *meta*, that optionally applies a series of *hooks* when initializing the metaclass

The hooks operate on the parent class of the metaclass, allowing the hook functions a method of dynamically modifying the parent class

Parameters **meta** : type

the metaclass that we will subclass

hooks : list of callables

functions taking a single argument (the class), which can be used to modify the class definition dynamically

Returns **wrapped** : metaclass

a subclass of *meta* that applies the specified hooks

class `nbodykit.plugins.`**`PluginBase`**(**args*, ***kwargs*)

Bases: `object`

A base class for plugins, designed to be subclassed to implement user plugins

The functionality here allows the plugins to be able to be loaded from YAML configuration files

Attributes

string	A unique identifier for the plugin, using the `id()`

Methods

create(plugin_name[, use_schema])	Instantiate a plugin from this extension type, based on the name/value pairs passed as keywo
fill_schema()	The class method responsible fill the class's schema with the relevant parameters from the _
from_config(parsed)	Instantiate a plugin from this extension point,

`__init__`(**args*, ***kwargs*)

classmethod **`create`**(*plugin_name*, *use_schema=False*, ***config*)

Instantiate a plugin from this extension type, based on the name/value pairs passed as keywords.

Optionally, cast the keywords values, using the types defined by the schema of the class we are creating

Parameters **plugin_name: str**

the name of the plugin to instantiate

use_schema : bool, optional

if *True*, cast the keywords that are defined in the class schema before initializing. Default: *False*

****config** : dict

the parameter names and values that will be passed to the plugin's __init__

Returns plugin :

the initialized instance of *plugin_name*

classmethod **`fill_schema`**()

The class method responsible fill the class's schema with the relevant parameters from the *`__init__()`* signature.

The schema allows the plugin to be initialized properly from a configuration file, validating the proper __init__ signatue.

This should call `add_argument()` of the class's `ConstructorSchema`, which is stored as the *`schema`* class attribute

classmethod **`from_config`**(*parsed*)

Instantiate a plugin from this extension point, based on the input *parsed* value, which is parsed directly from the YAML configuration file

There are several valid input cases for *parsed*:

1. parsed: dict containing the key *plugin*, which gives the name of the Plugin to load; the rest of the dictionary is treated as arguments of the Plugin
2. parsed: dict having only one entry, with key giving the Plugin name and value being a dictionary of arguments of the Plugin
3. parsed: dict if *from_config* is called directly from a Plugin class, then *parsed* can be a dictionary of the named arguments, with the Plugin name inferred from the class *cls*
4. parsed: str the name of a Plugin, which will be created with no arguments

`logger` **= <logging.Logger object>**

`string`

A unique identifier for the plugin, using the `id()` function

`nbodykit.plugins.`**`PluginBaseMeta`**

alias of `wrapped`

Submodules

nbodykit.plugins.fromfile module

exception `nbodykit.plugins.fromfile.`**`ConfigurationError`**

Bases: `Exception`

General exception for when parsing plugins from fails

exception `nbodykit.plugins.fromfile.`**`EmptyConfigurationError`**

Bases: *`nbodykit.plugins.fromfile.ConfigurationError`*

Specific parsing error when the YAML loader does not find any valid keys

exception `nbodykit.plugins.fromfile.`**`PluginParsingError`**

Bases: *`nbodykit.plugins.fromfile.ConfigurationError`*

Specific parsing error when the plugin fails to load from the configuration file

`nbodykit.plugins.fromfile.`**`ReadConfigFile`**(*stream*, *schema*)

Read parameters from a file using YAML syntax

The function uses the specified *schema* to:

- check if parameter values are consistent with *choices*

- infer the *type* of each parameter
- check if any required parameters are missing

Parameters **stream** : open file object, str

an open file object or the string returned by calling *read*

schema : ConstructorSchema

the schema which tells the parser which holds the relevant information about the necessary parameters

Returns **ns** : argparse.Namespace

the namespace holding the loaded parameters that are valid according to the input schema

unknown : argparse.Namespace

a namespace holding any parsed parameters not present in the scema

`nbodykit.plugins.fromfile.`**`case_insensitive_name_match`**(*schema_name*, *config*)

Do case-insensitive name matching between the ConstructorSchema and the parsed configuration file

Parameters **schema_name** : str

the name of the parameter, as given in the ConstructorSchema

config : dict

the parsed YAML dictionary from the configuration file

Returns **config_name** : {str, None}

return the key of *config* that matches *schema_name*; otherwise, return *None*

`nbodykit.plugins.fromfile.`**`fill_namespace`**(*ns*, *arg*, *config*, *missing*)

Recursively fill the input namespace from a dictionary parsed from a YAML configuration file

Parameters **ns** : argparse.Namespace

the namespace to fill with the loaded parameters

arg : Argument

the schema Argument instance that we are trying to add to the namespace; this holds the details about casting, sub-fields, etc

config : dict

a dictionary holding the parsed values from the YAML file

missing : list

a list to update with any arguments that are missing; i.e., required by the schema but not present in *config*

Notes

- Fields that have sub-fields will be returned as sub-namespaces, such that the subfields can be accessed from the parent field with the same `parent.subfield` syntax
- Comparison of names between and configuration file and schema are done in a case-insensitive manner
- Before adding to the namespace the values will be case according to the *cast* function specified via *arg*

nbodykit.plugins.hooks module

nbodykit.plugins.hooks.**add_and_validate_schema**(*cls*)

A hook that:

1. adds a ConstructorSchema to the input class
2. calls the `fill_schema()` class method, if available
3. decorates __init__ to validate arguments at the time of intialization

nbodykit.plugins.hooks.**add_logger**(*cls*)

A hook that adds a logger to the input class as a class attribute `logger`

nbodykit.plugins.hooks.**attach_comm**(*cls*)

A hook that attaches the *comm* keyword to a class. This class performs two operations:

1. it adds 'comm' to the class schema
2. it sets the 'comm' keyword argument, using the global value as the default

nbodykit.plugins.hooks.**attach_cosmo**(*cls*)

A hook that attaches the *cosmo* keyword to a class. This class performs two operations:

1. it adds 'cosmo' to the class schema
2. it sets the 'cosmo' keyword argument, using the global value as the default

nbodykit.plugins.manager module

class nbodykit.plugins.manager.**PluginManager**(*paths*, *qualprefix='nbodykit.core.user'*)

Bases: `object`

A class to manage the loading of plugins in `nbodykit`.

Note: This class uses the ingleton pattern, so only one instance exists when `nbodykit` is loaded

It should be accessed using the `get()` function.

Methods

`add_user_plugin`(*paths)	Dynamically load user plugins, adding each plugin
`create`(paths[, qualprefix])	Create the PluginManager instance
`format_help`(extension, *plugins)	Return a string specifying the *help* for each of the plugins
`get`()	Return the PluginManager
`get_plugin`(name[, type])	Return the plugin instance for the given plugin name

__init__(*paths*, *qualprefix='nbodykit.core.user'*)

Initialize a new PluginManager from the specified search paths

The user should use `get()` to get the instance of the PluginManager

add_user_plugin(**paths*)

Dynamically load user plugins, adding each plugin to `nbodykit.core.user`

Note: At the moment, loaded plugins must be a subclass of one of the classes defined in `supported_types`

Parameters **paths** : tuple of str

the file paths to search for plugins to load

classmethod **`create`**(*paths*, *qualprefix='nbodykit.core.user'*)

Create the PluginManager instance

Uses the singleton pattern to ensure that only one plugin manager exists

Parameters **paths** : tuple of str

the search paths to look for the core plugins

qualprefix : str, optional

the prefix to build a qualified name in `sys.modules`. This is used to load the builtin plugins in *`nbodykit.core`*

Raises **ValueError :**

if the PluginManager already exists

`format_help`(*extension*, **plugins*)

Return a string specifying the *help* for each of the plugins specified of extension type *extension*

If no *plugins* are specified, format the help message for all plugins registered as subclasses of *extension*

Parameters **extension** : str

the string specifying the extension type; should be in *`PluginManager.supported_types`*

plugins : list of str

strings specifying the names of plugins of the specified type to format together

Returns **help_msg** : str

the formatted help message string for the specified plugins

classmethod **`get`**()

Return the PluginManager

Raises **ValueError :**

if the PluginManager has not been created yet

`get_plugin`(*name*, *type=None*)

Return the plugin instance for the given plugin name

Parameters **name** : str

the name of the plugin to load

type : str, optional

the extension type (one of *`supported_types`*); in the case of name collisions across different extension types, this parameter must be given, otherwise, an exception will be raised

Returns cls

the plugin class corresponding to *name*

`supported_types` **= {'DataSource': <class 'nbodykit.core.datasource.DataSource'>, 'Painter': <class 'nbodykit.core.p**

nbodykit.plugins.schema module

class nbodykit.plugins.schema.**Argument**

Bases: *nbodykit.plugins.schema.Argument*

Class to represent an argument in the *ConstructorSchema*

Attributes

choices	Alias for field number 4
default	Alias for field number 3
help	Alias for field number 6
name	Alias for field number 0
nargs	Alias for field number 5
required	Alias for field number 1
subfields	Alias for field number 7
type	Alias for field number 2

Methods

count(...)	
index((value, [start, ...)	Raises ValueError if the value is not present.

nbodykit.plugins.schema.**ArgumentBase**

alias of *Argument*

class nbodykit.plugins.schema.**ConstructorSchema**(*description=''*)

Bases: collections.OrderedDict

A subclass of OrderedDict to hold *Argument* objects, with argument names as the keys of the dictionary.

Each *Argument* stores the relevant information of that argument, included *type*, *help*, *choices*, etc.

Each *Argument* also stores a *subfields* attribute, which is a new OrderedDict of *Argument* objects to store any sub-fields

Notes

You can test whether a full argument 'name' is in the schema with the *contains* function

>> argname = 'field.DataSource' >> schema.contains(argname) True

Arguments that are subfields can be accessed in a sequential dict-like fashion:

>> subarg = schema['field']['DataSource']

Methods

Argument	Class to represent an argument in the *ConstructorSchema*
add_argument(name[, type, default, choices, ...])	Add an argument to the schema
cast(arg, value)	Convenience function to cast values based on the *type* stored in *schema*.
clear(() -> None. Remove all items from od.)	

Continued on next page

Table 1.146 – continued from previous page

`contains`(key)	Check if the schema contains the full argument name, using
`copy`(() -> a shallow copy of od)	
`format_help`()	Return a string giving the help using the
`fromkeys`((S[, ...)	If not specified, the value defaults to None.
`get`((k[,d]) -> D[k] if k in D, ...)	
`items`	
`keys`	
`move_to_end`	Move an existing element to the end (or beginning if last==False).
`pop`((k[,d]) -> v, ...)	value. If key is not found, d is returned if given, otherwise KeyError
`popitem`(() -> (k, v), ...)	Pairs are returned in LIFO order if last is true or FIFO order if false.
`setdefault`((k[,d]) -> od.get(k,d), ...)	
`update`	
`values`	

class `Argument`

Bases: *`nbodykit.plugins.schema.Argument`*

Class to represent an argument in the *ConstructorSchema*

Attributes

`choices`	Alias for field number 4
`default`	Alias for field number 3
`help`	Alias for field number 6
`name`	Alias for field number 0
`nargs`	Alias for field number 5
`required`	Alias for field number 1
`subfields`	Alias for field number 7
`type`	Alias for field number 2

Methods

`count`(...)	
`index`((value, [start, ...)	Raises ValueError if the value is not present.

`ConstructorSchema.`**`__init__`**(*description=''*)

Initialize an empty schema, optionally passing a description of the schema

Parameters **description** : str, optional

the string description of the schema

`ConstructorSchema.`**`add_argument`**(*name, type=None, default=None, choices=None, nargs=None, help=None, required=False*)

Add an argument to the schema

Parameters **name** : str

the name of the parameter to add

type : callable, optional

a function that will cast the parsed value

default : optional

the default value for this parameter

choices : optional

the distinct values that the parameter can take

nargs : int, '*', '+', optional

the number of arguments that should be consumed for this parameter

help : str, optional

the help string

required : bool, optional

whether the parameter is required or not

static `ConstructorSchema.`**`cast`**`(`*arg*, *value*`)`

Convenience function to cast values based on the *type* stored in *schema*.

Note: If the *type* of *arg* is a tuple, then each type will be attempted in the order given.

Parameters **arg** : `Argument`

the *Argument* which gives the relevant meta-data to properly cast *value*

value :

the value we are casting, using the *type* attribute of *arg*

Returns casted :

the casted value; can have many types

`ConstructorSchema.`**`contains`**`(`*key*`)`

Check if the schema contains the full argument name, using . to represent subfields

Parameters **key** : str

the name of the argument to search for

Examples

>> argname = 'field.DataSource' >> schema.contains(argname) True

`ConstructorSchema.`**`format_help`**`()`

Return a string giving the help using the format preferred by the `numpy` documentation

`nbodykit.plugins.schema.`**`attribute`**`(`*name*, ***kwargs*`)`

A function decorator that adds an argument to the class's schema

See `ConstructorSchema.add_argument()` for further details

Parameters **name** : the name of the argument to add

****kwargs** : dict

the additional keyword values to pass to `add_argument`

nbodykit.plugins.validate module

`nbodykit.plugins.validate.`**`get_init_errmsg`**`(`*schema*, *posargs*, *kwargs*`)`

Return a reasonable error message, accounting for:

- missing arguments
- extra arguments
- duplicated positional + keyword arguments

`nbodykit.plugins.validate.`**`validate__init__`**`(`*init*`)`

Validate the input arguments to `__init__()` using the class schema

Parameters **init** : callable

the __init__ function we are decorating

`nbodykit.plugins.validate.`**`validate__init__signature`**`(`*func*, *attrs*, *defaults*`)`

Update the schema, which is attached to *func*, using information gathered from the function's signature, namely *attrs* and *defaults*

This will update the *required* and *default* values of the arguments of the schema, using the signature of *func*

It also verifies certain aspects of the schema, mostly as a consistency check for the developers

`nbodykit.plugins.validate.`**`validate_choices`**`(`*schema*, *args_dict*`)`

Verify that the input values are consistent with the *choices*, using the schema

`nbodykit.plugins.validate.`**`validate_required_attributes`**`(`*plugin*`)`

Validate that the plugin has the required attributes, where *plugin* has already been initialized

This looks for the :attr‘required_attributes‘ class attribute of plugin.

plugin [] the initialized plugin instance

nbodykit.utils package

General utility functions

`nbodykit.utils.`**`local_random_seed`**`(`*seed*, *comm*`)`

Return a random seed for the local rank, which is seeded by the global *seed*

Parameters **seed** : int, None

the global seed, used to seed the local random state

comm : MPI.Communicator

the MPI communicator

Returns int :

a integer appropriate for seeding the local random state

Notes

This attempts to avoid correlation between random states for different ranks by using the global seed to generate new seeds for each rank.

The seed must be a 32 bit unsigned integer, so it is selected between 0 and 4294967295

`nbodykit.utils.`**`timer`**`(`*start*, *end*`)`

Utility function to return a string representing the elapsed time, as computed from the input start and end times

Parameters **start** : int

the start time in seconds

end : int

the end time in seconds

Returns str :

the elapsed time as a string, using the format *hours:minutes:seconds*

Submodules

nbodykit.utils.meshtools module

class `nbodykit.utils.meshtools.`**`MeshSlab`**(*islab*, *coords*, *axis*, *symmetry_axis*)

Bases: `object`

A convenience class to represent a specific slab of a mesh, which is denoted as a `slab`

Attributes

hermitian_symmetric	Whether the slab is Hermitian-symmetric
hermitian_weights	Weights to be applied to quantities on the slab in order
index	Return an indexing list appropriate for selecting the slicing the
meshshape	Return the local shape of the mesh on this rank, as determined
nonsingular	Return the indices on the slab of the positive frequencies
shape	Return the shape of the slab

Methods

coords(i)	Return the coordinate array for dimension `i` on this slab,
mu(los_index)	The *mu* value defined at each point on the slab for the
norm2()	The square of coordinate grid norm defined at each point on the slab.

`__init__`(*islab*, *coords*, *axis*, *symmetry_axis*)

Parameters **islab** : int, [0, x[0].shape[0]]

the index of the slab, which indexes the first dimension of the mesh (the *x* coordinate), thus producing a y-z plane

coords : list of arrays

the coordinate arrays of the mesh, with proper 3D shapes for easy broadcasting; if the mesh has size (Nx, Ny, Nz), then the shapes of *x* are: [(Nx, 1, 1), (1, Ny, 1), (1, 1, Nz)]

axis : int, {0, 1, 2}

the index of the mesh axis to iterate over

symmetry_axis : int, optional

if provided, the axis that has been compressed due to Hermitian symmetry

coords(*i*)
Return the coordinate array for dimension `i` on this slab,

> **Note:** The return value will be properly squeezed for easy broadcasting, i.e., if *i* is *self.axis*, then an array of shape *(1,1)* array is returned, otherwise, the shape is *(N_i, 1)* or *(1, N_i)*

Parameters **i** : int, {0,1,2}

the index of the desired dimension

Returns array_like

the coordinate array for dimension *i* on the slab; see the note about the shape of the return array for details

hermitian_symmetric
Whether the slab is Hermitian-symmetric

hermitian_weights
Weights to be applied to quantities on the slab in order to account for Hermitian symmetry

These weights double-count the positive frequencies along the *symmetry_axis*.

index
Return an indexing list appropriate for selecting the slicing the appropriate slab out of a full 3D array of shape (Nx, Ny, Nz)

meshshape
Return the local shape of the mesh on this rank, as determined by the input coordinates array

mu(*los_index*)
The *mu* value defined at each point on the slab for the specified line-of-sight index

Parameters **los_index: int, {0, 1, 2}**

the index defining the line-of-sight, which *mu* is defined with respect to

Returns array_like, (slab.shape)

the *mu* value at each point in the slab

nonsingular
Return the indices on the slab of the positive frequencies along the dimension specified by *symmetry_axis*

norm2()
The square of coordinate grid norm defined at each point on the slab.

This broadcasts the coordinate arrays along each dimension to compute the norm at each point in the slab.

Returns array_like, (slab.shape)

the square of coordinate mesh at each point in the slab

shape
Return the shape of the slab

`nbodykit.utils.meshtools.`**SlabIterator**(*coords*, *axis=0*, *symmetry_axis=None*)
Iterate over the specified dimension of the coordinate mesh, returning a `MeshSlab` for each iteration

Parameters **coords** : list of arrays the coordinate arrays of the mesh, with proper 3D shapes for easy broadcasting; if the mesh has size (Nx, Ny, Nz), then the shapes of *x3d* should be: `[(Nx, 1, 1), (1, Ny, 1), (1, 1, Nz)]`

axis : int, optional

the index of the mesh axis to iterate over

symmetry_axis : int, optional

if provided, the axis that has been compressed due to Hermitian symmetry

nbodykit.utils.selectionlanguage module

class `nbodykit.utils.selectionlanguage.`**`BoolAnd`**(*t*)

Bases: *`nbodykit.utils.selectionlanguage.BoolBinOp`*

Methods

`eval`(*data)	

`eval`(**data*)

`reprsymbol` = **'&'**

class `nbodykit.utils.selectionlanguage.`**`BoolBinOp`**(*t*)

Bases: `object`

Methods

`eval`(*data)	

`__init__`(*t*)

`eval`(**data*)

class `nbodykit.utils.selectionlanguage.`**`BoolNot`**(*t*)

Bases: `object`

Methods

`eval`(*data)	

`__init__`(*t*)

`eval`(**data*)

class `nbodykit.utils.selectionlanguage.`**`BoolOr`**(*t*)

Bases: *`nbodykit.utils.selectionlanguage.BoolBinOp`*

Methods

eval(*data)	

eval(**data*)

reprsymbol = '|'

class nbodykit.utils.selectionlanguage.**ColumnSlice**(*r*)

Bases: object

The optional column index for the left-hand side of the query selection – this will be applied by LeftCompOperand, if present

Methods

isempty()	

__init__(*r*)

isempty()

class nbodykit.utils.selectionlanguage.**CompOperand**(*r*)

Bases: object

Base class for comparison operands

Methods

eval(data)	The main function that does the work for each operand, given

__init__(*r*)

Parameters r : ParseResults

the ParseResults instance; must have unity length

eval(*data*)

The main function that does the work for each operand, given input data array

Parameters data : array_like

array that has named fields, i.e., a structured array or DataFrame

class nbodykit.utils.selectionlanguage.**CompOperator**(*t*)

Bases: object

Class to parse the comparison operator and do the full comparison, joining the LeftCompOperand and RightCompOperand instances

Methods

eval(*data)	

`__init__`(*t*)

`eval`(**data*)

ops = {'<=': <built-in function le>, 'is not': <function CompOperator._isnot at 0x7fde5c27f620>, '!=': <built-in function

class `nbodykit.utils.selectionlanguage.`**`LeftCompOperand`**(*r*)

Bases: `nbodykit.utils.selectionlanguage.CompOperand`

The left operand that gives the name of the column. This class supports vector indexing syntax, similar to numpy.

This class is responsible for returning the specific column (possibly sliced) from the input data array

Examples

>> "LogMass < 12" >> "Position[:,0] < 1000.0" >> log10(Mass) < 12

Methods

`eval`(data)	Return the named column from the input data

`__init__`(*r*)

`eval`(*data*)

Return the named column from the input data

Parameters **data** : array_like

array that has named fields, i.e., a structured array or DataFrame

Returns array_like :

the specific column of the input data

class `nbodykit.utils.selectionlanguage.`**`Query`**(*str_selection*)

Bases: `object`

Class to parse boolean expressions and return boolean masks based on data with named fields.

Notes

- The string expression must be a *comparison condition*, separated by a boolean operator (*and*, *or*, *not*). A comparison condition has the syntax:

  column_name'|'*[index] comparison_operator value*

  column_name [] the name of a column in a data array. the values from the data array with this column name are substituted into the boolean expression

  comparison_operator : any of the following are valid: >, >=, <, <=, ==, !=, is, is not

  value [] This must be able to have *eval* called on it. Usually a number or single-quoted string; nan and inf are also supported.

- if *column_name* refers to a vector, the index of the vector can be passed as the index of the column, using the usual square bracket syntax
- *numpy.nan* and *numpy.inf* can be tested for by "is (not) nan" and "is (not) inf"

- As many *comparison conditions* as needed can be nested together, joined by *and*, *or*, or *not*
- *log*, *exp*, and *log10* are mapped to their numpy functions
- any of the above builtin functions can also be applied to the column names, i.e,, "log10(Mass) > 14"

Methods

`__call__`(data)	
`get_mask`(data)	Apply the selection to the specified data and return the
`parse_selection`(str_selection)	Parse the input string condition

`__init__`(*str_selection*)

Parameters **str_selection** : str

the boolean expression as a string

`get_mask`(*data*)

Apply the selection to the specified data and return the implied boolean mask

Parameters **data** : a dict like object

data object that must have named fields, necessary for type-casting the values in the selection string

Returns **mask** : list or array like

the boolean mask corresponding to the selection string

`parse_selection`(*str_selection*)

Parse the input string condition

class `nbodykit.utils.selectionlanguage.`**`RightCompOperand`**(*r*)

Bases: *`nbodykit.utils.selectionlanguage.CompOperand`*

The right operand that evaluates the comparison value.

This class is responsible for returning the evaluated value of the comparison key, i.e., a string, float, etc.

Note that *inf* and *nan* will be evaluated to their numpy counterparts.

Methods

`concat`(s)	Concatenate parsed results into one string for eval'ing purposes
`eval`(*args)	Evaluate the right side of the comparison using `eval`

`__init__`(*r*)

`concat`(*s*)

Concatenate parsed results into one string for eval'ing purposes

`eval`(**args*)

Evaluate the right side of the comparison using `eval`

exception `nbodykit.utils.selectionlanguage.`**`SelectionError`**

Bases: `Exception`

nbodykit.utils.taskmanager module

class `nbodykit.utils.taskmanager.`**`TaskManager`**(*task_function*, *cpus_per_worker*, *comm=None*, *debug=False*, *use_all_cpus=False*)

Bases: `object`

An MPI manager that distributes tasks over a set of MPI processes, using a specified number of independent workers to compute tasks

Given the specified number of independent workers (which compute tasks in parallel), the total number of available CPUs will be divided evenly.

The main function is `compute` which tasks a list of tasks and runs the task function for each item of the list

Methods

`compute`(tasks)	Compute a series of tasks.
`is_master`()	Is the current process the master?
`is_worker`()	Is the current process a valid worker (and thus should wait for
`wait`()	If this isn't the master process, wait for instructions.

`__init__`(*task_function*, *cpus_per_worker*, *comm=None*, *debug=False*, *use_all_cpus=False*)

Parameters **task_function** : callable

the task function which takes two arguments: the task iteration integer and the *task* value

cpus_per_worker : int

the desired number of ranks assigned to each independent worker in the pool

comm : MPI communicator, optional

the global communicator that will be split so each worker has a subset of CPUs available; default is COMM_WORLD

debug : bool, optional

if *True*, set the logging level to *DEBUG*, which prints out much more information; default is *False*

use_all_cpus : bool, optional

if *True*, use all available CPUs, including the remainder if *cpus_per_worker* is not divide the total number of CPUs evenly; default is *False*

`compute`(*tasks*)

Compute a series of tasks. For each task, the function takes the iteration number, followed by the *task* value as the arguments

Parameters **tasks** : list

list of *task* items that will be pickled and set to each worker when computing the *ith* task

Returns **results** : list

a list of the return values of the task function for each task

`is_master`()

Is the current process the master?

is_worker()

Is the current process a valid worker (and thus should wait for instructions from the master)

logger = <logging.Logger object>

wait()

If this isn't the master process, wait for instructions.

nbodykit.utils.taskmanager.**enum**(**sequential*, ***named*)

nbodykit.utils.taskmanager.**split_ranks**(*N_ranks*, *N*, *include_all=False*)

Divide the ranks into chunks, attempting to have *N* ranks in each chunk. This removes the master (0) rank, such that *N_ranks - 1* ranks are available to be grouped

Parameters **N_ranks** : int

the total number of ranks available

N : int

the desired number of ranks per worker

include_all : bool, optional

if *True*, then do not force each group to have exactly *N* ranks, instead including the remainder as well; default is *False*

Submodules

nbodykit.cosmology module

class nbodykit.cosmology.**Cosmology**(*H0=67.6*, *Om0=0.31*, *Ob0=0.0486*, *Ode0=0.69*, *w0=-1.0*, *Tcmb0=2.7255*, *Neff=3.04*, *m_nu=0.0*, *flat=False*)

Bases: nbodykit.cosmology.CosmologyBase

A class for computing cosmology-dependent quantites, which uses *astropy.Cosmology* to do the calculations

Notes

- this class supports the LambdaCDM and wCDM classes from *astropy* (and their flat equivalents)
- additions to the fiducial *LCDM* model include the dark energy equation of state *w0* and massive neutrinos
- if *flat* = *True*, the dark energy density is set automatically
- the underlying astropy class is stored as the *engine* attribute

Methods

comoving_distance(z)	Returns the comoving distance to z in units of *Mpc/h*
is_sampled(methodname)	
sample(methodname, x, *args, **kwargs)	
sampled_function(func, x, *args, **kwargs)	Class to represent a "sampled" version of a function
unsample(methodname)	

__init__(*H0=67.6*, *Om0=0.31*, *Ob0=0.0486*, *Ode0=0.69*, *w0=-1.0*, *Tcmb0=2.7255*, *Neff=3.04*, *m_nu=0.0*, *flat=False*)

comoving_distance(*z*)
Returns the comoving distance to z in units of *Mpc/h*

class nbodykit.cosmology.**CosmologyBase**
Bases: object

Base class for computing cosmology-dependent quantities, possibly sampling them at at set of points and returning interpolated results (for speed purposes)

Methods

is_sampled(methodname)	
sample(methodname, x, *args, **kwargs)	
sampled_function(func, x, *args, **kwargs)	Class to represent a "sampled" version of a function
unsample(methodname)	

is_sampled(*methodname*)

sample(*methodname*, *x*, **args*, ***kwargs*)

class sampled_function(*func*, *x*, **args*, ***kwargs*)
Bases: object

Class to represent a "sampled" version of a function

Methods

__call__(y)

__init__(*func*, *x*, **args*, ***kwargs*)

CosmologyBase.**unsample**(*methodname*)

nbodykit.cosmology.**neutrino_mass**(*value*)
Function to cast an input string or list to a *astropy.units.Quantity*, with units of *eV* to represent neutrino mass

nbodykit.dataset module

class nbodykit.dataset.**Corr1dDataSet**(*edges*, *variables*, ***kwargs*)
Bases: *nbodykit.dataset.DataSet*

A *DataSet* that holds a 1D correlation function in bins of *r*

Attributes

shape	The shape of the coordinate grid
variables	Alias to return the names of the variables stored in *data*

Methods

average(dim, **kwargs)	Compute the average of each variable over the specified dimension.
copy()	Returns a copy of the DataSet
from_nbkit(d, meta[, columns])	Return a *Corr1dDataSet* instance taking the return values
reindex(dim, spacing[, weights, force, ...])	Reindex the dimension *dim* by averaging over multiple coordinate bins, optionally w
rename_variable(old_name, new_name)	Rename a variable in data from *old_name* to *new_name*
sel([method])	Return a new DataSet indexed by coordinate values along the
squeeze([dim])	Squeeze the DataSet along the specified dimension, which

__init__(*edges*, *variables*, ***kwargs*)

classmethod from_nbkit(*d*, *meta*, *columns=None*, ***kwargs*)

Return a *Corr1dDataSet* instance taking the return values of *files.Read1DPlainText* as input

Parameters d : array_like

the 1D data stacked vertically, such that each columns represents a separate data variable

meta : dict

any additional metadata to store as part of the P(k) measurement

columns : list

list of the column names – required if *columns* not in *meta* dictionary

Examples

```
>>> from nbodykit import files
>>> corr = Power1dDataSet.from_nbkit(*files.Read1DPlainText(filename))
```

class nbodykit.dataset.**Corr2dDataSet**(*edges*, *variables*, ***kwargs*)

Bases: *nbodykit.dataset.DataSet*

A *DataSet* that holds a 2D correlation in bins of *k* and *mu*

Attributes

shape	The shape of the coordinate grid
variables	Alias to return the names of the variables stored in *data*

Methods

average(dim, **kwargs)	Compute the average of each variable over the specified dimension.
copy()	Returns a copy of the DataSet
from_nbkit(d, meta, **kwargs)	Return a *Corr2dDataSet* instance taking the return values
reindex(dim, spacing[, weights, force, ...])	Reindex the dimension *dim* by averaging over multiple coordinate bins, optionally w
rename_variable(old_name, new_name)	Rename a variable in data from *old_name* to *new_name*
sel([method])	Return a new DataSet indexed by coordinate values along the
squeeze([dim])	Squeeze the DataSet along the specified dimension, which

`__init__`(*edges*, *variables*, ***kwargs*)

classmethod **`from_nbkit`**(*d*, *meta*, ***kwargs*)

Return a *Corr2dDataSet* instance taking the return values of *files.Read2DPlainText* as input

Parameters **d** : dict

dictionary holding the *edges* data, as well as the data columns for the 2D measurement

meta : dict

any additional metadata to store as part of the 2D measurement

Examples

```
>>> from nbodykit import files
>>> corr = Corr2dDataSet.from_nbkit(*files.Read2DPlainText(filename))
```

class `nbodykit.dataset.`**`DataSet`**(*dims*, *edges*, *variables*, ***kwargs*)

Bases: `object`

Lightweight class to hold variables at fixed coordinates, i.e., a grid of (r, mu) or (k, mu) bins for a correlation function or power spectrum measurement

It is modeled after the syntax of `xarray.Dataset`, and is designed to hold correlation function or power spectrum results (in 1D or 2D)

Notes

- the suffix *cen* will be appended to the names of the dimensions passed to the constructor, since the `coords` array holds the **bin centers**, as constructed from the bin edges

Examples

The following example shows how to read a power spectrum or correlation function measurement as written by a nbodykit *Algorithm*. It uses *`Read1DPlainText()`*

```
>>> from nbodykit import files
>>> corr = Corr2dDataSet.from_nbkit(*files.Read2DPlainText(filename))
>>> pk = Power1dDataSet.from_nbkit(*files.Read1DPlainText(filename))
```

Data variables and coordinate arrays can be accessed in a dict-like fashion:

```
>>> power = pkmu['power'] # returns power data variable
>>> k_cen = pkmu['k_cen'] # returns k_cen coordinate array
```

Array-like indexing of a *`DataSet`* returns a new *`DataSet`* holding the sliced data:

```
>>> pkmu
<DataSet: dims: (k_cen: 200, mu_cen: 5), variables: ('mu', 'k', 'power')>
>>> pkmu[:,0] # select first mu column
<DataSet: dims: (k_cen: 200), variables: ('mu', 'k', 'power')>
```

Additional data variables can be added to the *`DataSet`* via:

```
>>> modes = numpy.ones((200, 5))
>>> pkmu['modes'] = modes
```

Coordinate-based indexing is possible through `sel()`:

```
>>> pkmu
<DataSet: dims: (k_cen: 200, mu_cen: 5), variables: ('mu', 'k', 'power')>
>>> pkmu.sel(k_cen=slice(0.1, 0.4), mu_cen=0.5)
<DataSet: dims: (k_cen: 30), variables: ('mu', 'k', 'power')>
```

`squeeze()` will explicitly squeeze the specified dimension (of length one) such that the resulting instance has one less dimension:

```
>>> pkmu
<DataSet: dims: (k_cen: 200, mu_cen: 1), variables: ('mu', 'k', 'power')>
>>> pkmu.squeeze(dim='mu_cen') # can also just call pkmu.squeeze()
<DataSet: dims: (k_cen: 200), variables: ('mu', 'k', 'power')>
```

`average()` returns a new `DataSet` holding the data averaged over one dimension

`reindex()` will re-bin the coordinate arrays along the specified dimension

Attributes

`shape`	The shape of the coordinate grid
`variables`	Alias to return the names of the variables stored in *data*

Methods

`average`(dim, **kwargs)	Compute the average of each variable over the specified dimension.
`copy`()	Returns a copy of the DataSet
`from_nbkit`(d, meta)	Return a DataSet object from a dictionary of data and
`reindex`(dim, spacing[, weights, force, ...])	Reindex the dimension *dim* by averaging over multiple coordinate bins, optionally we
`rename_variable`(old_name, new_name)	Rename a variable in `data` from *old_name* to *new_name*
`sel`([method])	Return a new DataSet indexed by coordinate values along the
`squeeze`([dim])	Squeeze the DataSet along the specified dimension, which

`__init__`(*dims*, *edges*, *variables*, ***kwargs*)

Parameters **dims** : list, (Ndim,)

A list of strings specifying names for the coordinate dimensions. The dimension names stored in `dims` have the suffix 'cen' added, to indicate that the coordinate grid is defined at the bin centers

edges : list, (Ndim,)

A list specifying the bin edges for each dimension

variables : dict

a dictionary holding the data variables, where the keys are interpreted as the variable names. The variable names are stored in `variables`

****kwargs :**

Any additional keywords are saved as metadata in the `attrs` attribute, which is an `OrderedDict`

`average`(*dim*, ***kwargs*)
Compute the average of each variable over the specified dimension.

Parameters **dim** : str

The name of the dimension to average over

****kwargs :**

Additional keywords to pass to *`DataSet.reindex()`*. See the documentation for *`DataSet.reindex()`* for valid keywords.

Returns **averaged** : DataSet

A new DataSet, with data averaged along one dimension, which reduces the number of dimension by one

`copy`()
Returns a copy of the DataSet

classmethod **`from_nbkit`**(*d*, *meta*)
Return a DataSet object from a dictionary of data and metadata

Parameters **d** : dict

A dictionary holding the data variables, as well as the *dims* and *edges* values

meta : dict

dictionary of metadata to store in the `attrs` attribute

Returns DataSet

The newly constructed DataSet

Notes

- The dictionary *d* must also have entries for *dims* and *edges* which are used to construct the DataSet

`reindex`(*dim*, *spacing*, *weights=None*, *force=True*, *return_spacing=False*, *fields_to_sum=[]*)
Reindex the dimension *dim* by averaging over multiple coordinate bins, optionally weighting by *weights*. Return a new DataSet holding the re-binned data

Parameters **dim** : str

The name of the dimension to average over

spacing : float

The desired spacing for the re-binned data. If *force = True*, the spacing used will be the closest value to this value, such that the new bins are N times larger, when N is an integer

weights : array_like or str, optional (*None*)

An array to weight the data by before re-binning, or if a string is provided, the name of a data column to use as weights

force : bool, optional

If *True*, force the spacing to be a value such that the new bins are N times larger, when N is an integer, otherwise, raise an exception. Default is *True*

return_spacing : bool, optional

If *True*, return the new spacing as the second return value. Default is *False*.

fields_to_sum : list

the name of fields that will be summed when reindexing, instead of averaging

Returns rebinned : DataSet

A new DataSet instance, which holds the rebinned coordinate grid and data variables

spacing : float, optional

If *return_spacing* is *True*, the new coordinate spacing will be returned

Notes

- We can only re-bin to an integral factor of the current dimension size in order to inaccuracies when re-binning to overlapping bins
- Variables specified in *fields_to_sum* will be summed when re-indexing, instead of averaging

`rename_variable`(*old_name*, *new_name*)

Rename a variable in `data` from *old_name* to *new_name*

Note that this procedure is performed in-place (does not return a new DataSet)

Parameters old_name : str

the name of the old varibale to rename

new_name : str

the desired new variable name

Raises ValueError

If *old_name* is not present in `variables`

`sel`(*method=None*, ***indexers*)

Return a new DataSet indexed by coordinate values along the specified dimension(s)

Parameters method : {None, 'nearest'}

The method to use for inexact matches; if set to *None*, require an exact coordinate match, otherwise match the nearest coordinate

****indexers :**

the pairs of dimension name and coordinate value used to index the DataSet

Returns sliced : DataSet

a new DataSet holding the sliced data and coordinate grid

Notes

Scalar values used to index a specific dimension will result in that dimension being squeezed. To keep a dimension of unit length, use a list to index (see examples below).

Examples

```
>>> pkmu
<DataSet: dims: (k_cen: 200, mu_cen: 5), variables: ('mu', 'k', 'power')>
```

```
>>> pkmu.sel(k_cen=0.4)
<DataSet: dims: (mu_cen: 5), variables: ('mu', 'k', 'power')>
```

```
>>> pkmu.sel(k_cen=[0.4])
<DataSet: dims: (k_cen: 1, mu_cen: 5), variables: ('mu', 'k', 'power')>
```

```
>>> pkmu.sel(k_cen=slice(0.1, 0.4), mu_cen=0.5)
<DataSet: dims: (k_cen: 30), variables: ('mu', 'k', 'power')>
```

shape
The shape of the coordinate grid

squeeze (*dim=None*)
Squeeze the DataSet along the specified dimension, which removes that dimension from the DataSet

The behavior is similar to that of `numpy.squeeze()`.

Parameters **dim** : str, optional

The name of the dimension to squeeze. If no dimension is provided, then the one dimension with unit length will be squeezed

Returns **squeezed** : DataSet

a new DataSet instance, squeezed along one dimension

Raises **ValueError**

If the specified dimension does not have length one, or no dimension is specified and multiple dimensions have length one

Examples

```
>>> pkmu
<DataSet: dims: (k_cen: 200, mu_cen: 1), variables: ('mu', 'k', 'power')>
>>> pkmu.squeeze() # squeeze the mu dimension
<DataSet: dims: (k_cen: 200), variables: ('mu', 'k', 'power')>
```

variables
Alias to return the names of the variables stored in *data*

class nbodykit.dataset.**Power1dDataSet** (*edges*, *variables*, ***kwargs*)
Bases: *nbodykit.dataset.DataSet*

A *DataSet* that holds a 1D power spectrum in bins of *k*

Attributes

shape	The shape of the coordinate grid
variables	Alias to return the names of the variables stored in *data*

Methods

`average`(dim, **kwargs)	Compute the average of each variable over the specified dimension.
`copy`()	Returns a copy of the DataSet
`from_nbkit`(d, meta[, columns])	Return a *Power1dDataSet* instance taking the return values
`reindex`(dim, spacing[, weights, force, ...])	Reindex the dimension *dim* by averaging over multiple coordinate bins, optionally w
`rename_variable`(old_name, new_name)	Rename a variable in `data` from *old_name* to *new_name*
`sel`([method])	Return a new DataSet indexed by coordinate values along the
`squeeze`([dim])	Squeeze the DataSet along the specified dimension, which

`__init__`(*edges*, *variables*, ***kwargs*)

classmethod `from_nbkit`(*d*, *meta*, *columns=None*, ***kwargs*)
Return a *Power1dDataSet* instance taking the return values of *files.Read1DPlainText* as input

Parameters **d** : array_like

the 1D data stacked vertically, such that each columns represents a separate data variable

meta : dict

any additional metadata to store as part of the P(k) measurement

columns : list

list of the column names – required if *columns* not in *meta* dictionary

Examples

```
>>> from nbodykit import files
>>> power = Power1dDataSet.from_nbkit(*files.Read1DPlainText(filename))
```

class `nbodykit.dataset.`**`Power2dDataSet`**(*edges*, *variables*, ***kwargs*)
Bases: `nbodykit.dataset.DataSet`

A *DataSet* that holds a 2D power spectrum in bins of *k* and *mu*

Attributes

`shape`	The shape of the coordinate grid
`variables`	Alias to return the names of the variables stored in *data*

Methods

`average`(dim, **kwargs)	Compute the average of each variable over the specified dimension.
`copy`()	Returns a copy of the DataSet
`from_nbkit`(d, meta, **kwargs)	Return a *Power2dDataSet* instance taking the return values
`reindex`(dim, spacing[, weights, force, ...])	Reindex the dimension *dim* by averaging over multiple coordinate bins, optionally w
`rename_variable`(old_name, new_name)	Rename a variable in `data` from *old_name* to *new_name*
`sel`([method])	Return a new DataSet indexed by coordinate values along the

Contir

Table 1.174 – continued from previous page

squeeze([dim])	Squeeze the DataSet along the specified dimension, which

__init__(*edges*, *variables*, ***kwargs*)

classmethod from_nbkit(*d*, *meta*, ***kwargs*)
Return a *Power2dDataSet* instance taking the return values of *files.Read2DPlainText* as input

Parameters **d** : dict

dictionary holding the *edges* data, as well as the data columns for the 2D measurement

meta : dict

any additional metadata to store as part of the 2D measurement

Examples

```
>>> from nbodykit import files
>>> power = Power2dDataSet.from_nbkit(*files.Read2DPlainText(filename))
```

nbodykit.dataset.**bin_ndarray**(*ndarray*, *new_shape*, *weights=None*, *operation=<function mean>*)
Bins an ndarray in all axes based on the target shape, by summing or averaging.

Parameters **ndarray** : array_like

the input array to re-bin

new_shape : tuple

the tuple holding the desired new shape

weights : array_like, optional

weights to multiply the input array by, before running the re-binning operation,

Notes

- Dimensions in *new_shape* must be integral factor smaller than the old shape
- Number of output dimensions must match number of input dimensions.
- See https://gist.github.com/derricw/95eab740e1b08b78c03f

Examples

```
>>> m = numpy.arange(0,100,1).reshape((10,10))
>>> n = bin_ndarray(m, new_shape=(5,5), operation=numpy.sum)
>>> print(n)
[[ 22  30  38  46  54]
 [102 110 118 126 134]
 [182 190 198 206 214]
 [262 270 278 286 294]
 [342 350 358 366 374]]
```

nbodykit.dataset.**from_1d_measurement**(*dims*, *d*, *meta*, *columns=None*, ***kwargs*)
Return a DataSet object from columns of data and any additional meta data

nbodykit.dataset.**from_2d_measurement**(*dims*, *d*, *meta*, ***kwargs*)

Return a DataSet object from a dictionary of data and any additional data

nbodykit.distributedarray module

class nbodykit.distributedarray.**DistributedArray**(*local*, *comm=<mpi4py.MPI.Intracomm object>*)

Bases: `object`

Distributed Array Object

A distributed array is striped along ranks

Attributes

comm	(`mpi4py.MPI.Comm`) the communicator
local	(array_like) the local data

Methods

bincount([local])	Assign count numbers from sorted local data.
sort([orderby])	Sort array globally by key orderby.
unique_labels()	Assign unique labels to sorted local.

__init__(*local*, *comm=<mpi4py.MPI.Intracomm object>*)

bincount(*local=False*)

Assign count numbers from sorted local data.

Warning: local data must be sorted, and of integer type. (numpy.bincount)

Parameters local : boolean

if local is True, only count the local array.

Returns N : *DistributedArray*

distributed counts array. If items of the same value spans other chunks of array, they are added to N as well.

Examples

if the local array is [(0, 0), (0, 1)], Then the counts array is [(3,), (3, 1)]

sort(*orderby=None*)

Sort array globally by key orderby.

Due to a limitation of mpsort, self[orderby] must be u8.

unique_labels()

Assign unique labels to sorted local.

Warning: local data must be sorted, and of simple type. (numpy.unique)

Returns **label** : *DistributedArray*

the new labels, starting from 0

class `nbodykit.distributedarray.`**`EmptyRankType`**

Bases: `object`

`nbodykit.distributedarray.`**`GatherArray`**(*data*, *comm*, *root=0*)

Gather the input data array from all ranks to the specified `root`

This uses *Gatherv*, which avoids mpi4py pickling, and also avoids the 2 GB mpi4py limit for bytes using a custom datatype

Parameters **data** : array_like

the data on each rank to gather

comm : MPI communicator

the MPI communicator

root : int

the rank number to gather the data to

Returns **recvbuffer** : array_like, None

the gathered data on root, and *None* otherwise

class `nbodykit.distributedarray.`**`LinearTopology`**(*local*, *comm*)

Bases: `object`

Helper object for the topology of a distributed array

Methods

`heads`()	The first items on each rank.
`next`()	The item after the local data.
`prev`()	The item before the local data.
`tails`()	The last items on each rank.

`__init__`(*local*, *comm*)

`heads`()

The first items on each rank.

Returns **heads** : list

a list of first items, EmptyRank is used for empty ranks

`next`()

The item after the local data.

This method the first item after the local data. If the rank after current rank is empty, item after that rank is used.

If no item is after local data, EmptyRank is returned.

Returns **next** : scalar

Item after local data, or EmptyRank if all ranks after this rank is empty.

prev()

The item before the local data.

This method fetches the last item before the local data. If the rank before is empty, the rank before is used.

If no item is before this rank, EmptyRank is returned

Returns **prev** : scalar

Item before local data, or EmptyRank if all ranks before this rank is empty.

tails()

The last items on each rank.

Returns tails: list

a list of last items, EmptyRank is used for empty ranks

nbodykit.distributedarray.**ScatterArray**(*data*, *comm*, *root=0*)

Scatter the input data array across all ranks, assuming *data* is initially only on *root* (and *None* on other ranks)

This uses *Scatterv*, which avoids mpi4py pickling, and also avoids the 2 GB mpi4py limit for bytes using a custom datatype

Parameters **data** : array_like or None

on *root*, this gives the data to split and scatter

comm : MPI communicator

the MPI communicator

root : int

the rank number that initially has the data

Returns **recvbuffer** : array_like

the chunk of *data* that each rank gets

nbodykit.distributedarray.**test**()

nbodykit.files module

class nbodykit.files.**GadgetGroupTabFile**(*basename*, *fid*, *args={'veldtype': 'f4', 'massdtype': 'f4', 'posdtype': 'f8'}*)

Bases: *nbodykit.stripedfile.StripeFile*

Methods

enum(filetype, basename[, args])	Iterate over all files of the type
read(column[, mystart, myend])	
readat(offset, nitem, dtype)	
write(column, mystart, data)	
writeat(offset, data)	

__init__(*basename*, *fid*, *args={'veldtype': 'f4', 'massdtype': 'f4', 'posdtype': 'f8'}*)

read(*column*, *mystart=0*, *myend=-1*)

class `nbodykit.files.`**`GadgetSnapshotFile`**(*basename*, *fid*, *args={'veldtype': 'f4', 'ptype': 1, 'massdtype': 'f4', 'iddtype': 'u8', 'posdtype': 'f8'}*)

Bases: `nbodykit.stripedfile.StripeFile`

Methods

`enum`(filetype, basename[, args])	Iterate over all files of the type
`read`(column[, mystart, myend])	
`readat`(offset, nitem, dtype)	
`write`(column, mystart, data)	
`writeat`(offset, data)	

`__init__`(*basename*, *fid*, *args={'veldtype': 'f4', 'ptype': 1, 'massdtype': 'f4', 'iddtype': 'u8', 'posdtype': 'f8'}*)

`read`(*column*, *mystart=0*, *myend=-1*)

class `nbodykit.files.`**`HaloLabelFile`**(*filename*, *fid*, *args*)

Bases: `nbodykit.stripedfile.StripeFile`

nbodykit halo label file

Attributes

npart	(int) Number of particles in this file
linking_length	(float) linking length. For example, 0.2 or 0.168

Methods

`enum`(filetype, basename[, args])	Iterate over all files of the type
`read`(column, mystart, myend)	Read a property column of particles
`readat`(offset, nitem, dtype)	
`write`(column, mystart, data)	
`writeat`(offset, data)	

`__init__`(*filename*, *fid*, *args*)

`nbodykit.files.`**`Read1DPlainText`**(*filename*)

Reads the plain text storage of a 1D measurement, as output by the *nbodykit.plugins.Measurement1DStorage* plugin.

Returns **data** : array_like

the 1D data stacked vertically, such that each columns represents a separate data variable

metadata : dict

any additional metadata to store as part of the P(k) measurement

Notes

- If *edges* is present in the file, they will be returned as part of the metadata, with the key *edges*
- If the first line of the file specifies column names, they will be returned as part of the metadata with the *columns* key

`nbodykit.files.`**`Read2DPlainText`**(*filename*)

Reads the plain text storage of a 2D measurement, as output by the *nbodykit.plugins.Measurement2DStorage* plugin

Returns data : dict

dictionary holding the *edges* data, as well as the data columns for the P(k,mu) measurement

metadata : dict

any additional metadata to store as part of the P(k,mu) measurement

class `nbodykit.files.`**`Snapshot`**(*filename*, *filetype*)

Bases: `object`

Methods

create(kls, filename, filetype, npart)	create a striped snapshot.
get_file(i)	
read(column, start, end)	this function provides a continuous view of multiple files
write(column, start, data)	this function provides a continuous view of multiple files

`__init__`(*filename*, *filetype*)

classmethod `create`(*kls*, *filename*, *filetype*, *npart*)

create a striped snapshot. npart is a list of npart for each file

`get_file`(*i*)

`read`(*column*, *start*, *end*)

this function provides a continuous view of multiple files

`write`(*column*, *start*, *data*)

this function provides a continuous view of multiple files

class `nbodykit.files.`**`TPMSnapshotFile`**(*basename*, *fid*, *args={}*)

Bases: *`nbodykit.stripedfile.StripeFile`*

Methods

create(kls, basename, fid, npart[, meta])	
`enum`(filetype, basename[, args])	Iterate over all files of the type
read(column[, mystart, myend])	
`readat`(offset, nitem, dtype)	
write(column, mystart, data)	
`writeat`(offset, data)	

__init__(*basename*, *fid*, *args={}*)

classmethod **create**(*kls*, *basename*, *fid*, *npart*, *meta={}*)

read(*column*, *mystart=0*, *myend=-1*)

write(*column*, *mystart*, *data*)

nbodykit.fkp module

class nbodykit.fkp.**FKPCatalog**(*data*, *randoms*, *BoxSize=None*, *BoxPad=0.02*, *compute_fkp_weights=False*, *P0_fkp=None*, *nbar=None*, *fsky=None*)

Bases: `object`

A *DataSource* representing a catalog of tracer objects, designed to be used in analysis similar to that first outlined by Feldman, Kaiser, and Peacock (FKP) 1994 (astro-ph/9304022)

In particular, the *FKPCatalog* uses a catalog of random objects to define the mean density of the survey, in addition to the catalog of data objects.

Attributes

data	Explicitly keep track of the *data* DataSource, in order to track

randoms	(DataSource) a *DataSource* that returns the position, weight, etc of a catalog of objects generated randomly to match the survey geometry and whose instrinsic clustering is zero
BoxSize	(array_like (3,)) the size of the cartesian box – the Cartesian coordinates of the input objects are computed using the input cosmology, and then placed into the box
mean_coordinate_offset	(array_like, (3,)) the average coordinate value in each dimension – this offset is used to return cartesian coordinates translated into the domain of [-BoxSize/2, BoxSize/2]

Methods

close()	Close an 'open' FKPCatalog
open()	The FKPCatalog class is designed to be used as a context manager, and this function serves to 'open'
paint(pm[, paintbrush])	Paint the FKP weighted density field: `data - alpha*randoms` using
read(name, columns[, full])	Read columns from either *data* or `randoms`, which is

__init__(*data*, *randoms*, *BoxSize=None*, *BoxPad=0.02*, *compute_fkp_weights=False*, *P0_fkp=None*, *nbar=None*, *fsky=None*)

Parameters **data** : DataSource

the DataSource that corresponds to the 'data' catalog

randoms : DataSource

the DataSource that corresponds to the 'randoms' catalog

BoxSize : {float, array_like (3,)}, optional

the size of the Cartesian to box when gridding the data and randoms; if not provided, the box will automatically be computed from the maximum extent of the randoms catalog

BoxPad : float, optional

if BoxSize is not provided, apply this padding to the box that is automatically created from the randoms catalog; default is `0.02`

compute_fkp_weights : bool, optional

if `True`, compute and apply FKP weights using *P0_fkp* and the number density n(z) column from the input data sources; default is `False`

P0_fkp : float, optional

if `compute_fkp_weights=True`, use this value in in the FKP weights, which are defined as, $w_\mathrm{FKP} = 1/(1+n_g(x)*P_0)$

nbar : {str, float}, optional

either the name of a file, giving `(z, n(z))` in columns, or a scalar float, which gives the constant n(z) value to use

fsky : float, optional

if `nbar = None`, then the n(z) is computed from the randoms catalog, and the sky fraction covered by the survey is required to properly normalize the number density (for the volume calculation)

`alpha`

Return the ratio of the data number density to the randoms number density

This is computed using the "completeness weights" :

$$\alpha = \sum_\mathrm{gal} w_{c,i} / \sum_\mathrm{ran} w_{c,i}$$

where $w_{c,i}$ is the completeness weight for the ith galaxy. This is specified via the `Weight` column, and has a default value of 1.

`close``()`

Close an 'open' FKPCatalog

This performs the following actions:

1. close the streams of the *`data`* and `randoms` DataSource objects
2. delete the stream attributes, so any cache of the DataSource objects will be automatically deleted, to free up memory

`closed`

Return *True* if the catalog has been setup and the data and random streams are open

`data`

Explicitly keep track of the *data* DataSource, in order to track the total number of objects

`default_columns`

Return a dictionary of the default columns to be read from the *`data`* and `randoms` attributes.

`fsky`

The sky area fraction (relative to 4π steradians)

Note: If $n(z)$ is not provided, then it will be computed from the randoms catalog. This attribute is needed for the volume normalization in that calculation, and must be set; otherwise, the code will crash

nbar
A callable function that returns the number density `n(z)` as a function of redshift (provided via argument)

open()
The FKPCatalog class is designed to be used as a context manager, and this function serves to 'open' the catalog.

This function performs the following actions:

- open the streams for the *data* and `randoms` DataSources, which allows them to be read from
- if needed, compute the Cartesian BoxSize using the maximum extent of the Cartesian coordinates of the `randoms` DataSource
- compute the `mean_coordinate_offset` from the `randoms` DataSource, which is used to re-center the Cartesian coordinates of the *data* and `randoms` to the range of `[-BoxSize/2, BoxSize/2]`
- if *nbar* is `None`, compute the $n(z)$ for the `randoms`, using the `RedshiftHistogramAlgorithm`

The desired usage for this function is

```
# initialize the catalog
catalog = FKPCatalog(data, randoms, **kwargs)

# open the catalog
with catalog:
    ...
```

In the above snippet, the `with` statement automatically calls the `open` function.

paint(*pm*, *paintbrush='cic'*)
Paint the FKP weighted density field: `data - alpha*randoms` using the input *ParticleMesh*

The are two different weights that enter into the painting procedure:

1. **completeness weights**: these weight each number density field for data and randoms seperately
2. **FKP weights**: these weight the total FKP density field, i.e., they weight `data - alpha*randoms`

Parameters **pm** : ParticleMesh

the particle mesh instance to paint the density field to

Returns **stats** : dict

a dictionary of FKP statistics, including total number, normalization, and shot noise parameters (see equations 13-15 of Beutler et al. 2013)

read(*name*, *columns*, *full=False*)
Read columns from either *data* or `randoms`, which is specified by the *name* argument

Parameters **name** : {'data', 'randoms'}

which DataSource to read the columns from

columns : list

the list of the names of the columns to read

full : bool, optional

if *True*, ignore any *bunchsize* parameters when reading from the specified DataSource

Returns list of arrays

the list of the data arrays for each column in `columns`

`nbodykit.fkp.`**`is_float`**`(`*s*`)`

Determine if a string can be cast safely to a float

nbodykit.fof module

`nbodykit.fof.`**`fof`**`(`*datasource*, *linking_length*, *nmin*, *comm=<mpi4py.MPI.Intracomm object>*, *log_level=10*`)`

Run Friend-of-friend halo finder.

Friend-of-friend was first used by Davis et al 1985 to define halos in hierachical structure formation of cosmological simulations. The algorithm is also known as DBSCAN in computer science. The subroutine here implements a parallel version of the FOF.

The underlying local FOF algorithm is from *kdcount.cluster*, which is an adaptation of the implementation in Volker Springel's Gadget and Martin White's PM. It could have been done faster.

Parameters **datasource: DataSource**

datasource; must support Position. datasource.BoxSize is used too.

linking_length: float

linking length in data units. (Usually Mpc/h).

nmin: int

Minimal length (number of particles) of a halo. Features with less than nmin particles are considered noise, and removed from the catalogue

comm: MPI.Comm

The mpi communicator.

Returns label: array_like

The halo label of each position. A label of 0 standands for not in any halo.

`nbodykit.fof.`**`fof_catalogue`**`(`*datasource*, *label*, *comm*, *calculate_initial=False*`)`

Catalogue of FOF groups based on label from a data source

Friend-of-friend was first used by Davis et al 1985 to define halos in hierachical structure formation of cosmological simulations. The algorithm is also known as DBSCAN in computer science. The subroutine here implements a parallel version of the FOF.

The underlying local FOF algorithm is from *kdcount.cluster*, which is an adaptation of the implementation in Volker Springel's Gadget and Martin White's PM. It could have been done faster.

Parameters **label** : array_like

halo label of particles from data source.

datasource: DataSource

datasource; must support Position and Velocity. datasource.BoxSize is used too.

comm: MPI.Comm

The mpi communicator. Must agree with the datasource

Returns catalogue: array_like

A 1-d array of type 'Position', 'Velocity', 'Length'. The center mass position and velocity of the FOF halo, and Length is the number of particles in a halo. The catalogue is sorted such that the most massive halo is first. catalogue[0] does not correspond to any halo.

`nbodykit.fof.`**`fof_halo_label`**(*minid*, *comm*, *thresh*)

Convert minid to sequential labels starting from 0.

This routine is used to assign halo label to particles with the same minid. Halos with less than thresh particles are reclassified to 0.

Parameters **minid** : array_like, ('i8')

The minimum particle id of the halo. All particles of a halo have the same minid

thresh : int

halo with less than thresh particles are merged into halo 0

comm : py:class:*MPI.Comm*

communicator. since this is a collective operation

Returns **labels** : array_like ('i8')

The new labels of particles. Note that this is ordered by the size of halo, with the exception 0 represents all particles that are in halos that contain less than thresh particles.

`nbodykit.fof.`**`fof_merge`**(*layout*, *minid*, *comm*)

`nbodykit.fof.`**`local_fof`**(*layout*, *pos*, *boxsize*, *ll*, *comm*)

`nbodykit.fof.`**`split_size_3d`**(*s*)

Split *s* into two integers, a and d, such that a * d == s and a <= d

returns: a, d

nbodykit.halos module

`nbodykit.halos.`**`centerofmass`**(*label*, *pos*, *boxsize=1.0*, *comm=<mpi4py.MPI.Intracomm object>*)

Calulate the center of mass of particles of the same label.

The center of mass is defined as the mean of positions of particles, but care has to be taken regarding to the periodic boundary.

This is a collective operation, and after the call, all ranks will have the position of halos.

Parameters **label** : array_like (integers)

Halo label of particles, >=0

pos : array_like (float, 3)

position of particles.

boxsize : float or None

size of the periodic box, or None if no periodic boundary is assumed.

comm : `MPI.Comm`

communicator for the collective operation.

Returns **hpos** : array_like (float, 3)

the center of mass position of the halos.

`nbodykit.halos.`**`count`**(*label*, *comm=<mpi4py.MPI.Intracomm object>*)

Count the number of particles of the same label.

This is a collective operation, and after the call, all ranks will have the particle count.

Parameters **label** : array_like (integers)

Halo label of particles, >=0

comm : `MPI.Comm`

communicator for the collective operation.

Returns **count** : array_like

the count of number of particles in each halo

nbodykit.measurestats module

`nbodykit.measurestats.`**`apply_bianchi_kernel`**(*data*, *x3d*, *i*, *j*, *k=None*)

Apply coordinate kernels to `data` necessary to compute the power spectrum multipoles via FFTs using the algorithm detailed in Bianchi et al. 2015.

This multiplies by one of two kernels:

1.x_i * x_j / x**2 * data, if *k* is None

2.x_i**2 * x_j * x_k / x**4 * data, if *k* is not None

See equations 10 (for quadrupole) and 12 (for hexadecapole) of Bianchi et al 2015.

Parameters **data** : array_like

the array to rescale – either the configuration-space *pm.real* or the Fourier-space *pm.complex*

x : array_like

the coordinate array – either *pm.r* or *pm.k*

i, j, k : int

the integers specifying the coordinate axes; see the above description

`nbodykit.measurestats.`**`compute_3d_corr`**(*fields*, *pm*, *comm=None*)

Compute the 3D correlation function by Fourier transforming the 3D power spectrum.

See the documentation for `compute_3d_power()` for details of input parameters and return types.

`nbodykit.measurestats.`**`compute_3d_power`**(*fields*, *pm*, *comm=None*)

Compute and return the 3D power from two input fields

Parameters **fields** : list of tuples of (DataSource, Painter, Transfer)

the list of fields which the 3D power will be computed

pm : ParticleMesh

the particle mesh object that handles the painting and FFTs

comm : MPI.Communicator, optional

the communicator to pass to the ParticleMesh object. If not provided, `MPI.COMM_WORLD` is used

Returns p3d : array_like (complex)

the 3D complex array holding the power spectrum

stats1 : dict

statistics of the first field, as returned by the *Painter*

stats2 : dict

statistics of the second field, as returned by the *Painter*

`nbodykit.measurestats.`**`compute_bianchi_poles`**(*comm*, *max_ell*, *catalog*, *Nmesh*, *factor_hexadecapole=False*, *paintbrush='cic'*)

Use the algorithm detailed in Bianchi et al. 2015 to compute and return the 3D power spectrum multipoles (*ell = [0, 2, 4]*) from one input field, which contains non-trivial survey geometry.

The estimator uses the FFT algorithm outlined in Bianchi et al. 2015 (http://adsabs.harvard.edu/abs/2015arXiv150505341B) to compute the monopole, quadrupole, and hexadecapole

Parameters comm : MPI.Communicator

the communicator to pass to the ParticleMesh object

max_ell : int, {0, 2, 4}

the maximum multipole number to compute up to (inclusive)

catalog : *`FKPCatalog`*

the FKP catalog object that manipulates the data and randoms DataSource objects and paints the FKP density

Nmesh : int

the number of cells (per side) in the gridded mesh

factor_hexadecapole : bool, optional

if *True*, use the factored expression for the hexadecapole (ell=4) from eq. 27 of Scoccimarro 2015 (1506.02729); default is *False*

paintbrush : str, {'cic', 'tsc'}

the density assignment kernel to use when painting, either *cic* (2nd order) or *tsc* (3rd order); default is *cic*

Returns pm : ParticleMesh

the mesh object used to do painting, FFTs, etc

result : list of arrays

list of 3D complex arrays holding power spectrum multipoles; respectively, if *ell_max=0,2,4*, the list holds the monopole only, monopole and quadrupole, or the monopole, quadrupole, and hexadecapole

stats : dict

dict holding the statistics of the input fields, as returned by the *FKPPainter* painter

References

- Bianchi, Davide et al., *Measuring line-of-sight-dependent Fourier-space clustering using FFTs*, MNRAS, 2015
- Scoccimarro, Roman, *Fast estimators for redshift-space clustering*, Phys. Review D, 2015

`nbodykit.measurestats.`**`compute_brutal_corr`**(*datasources*, *redges*, *Nmu=0*, *comm=None*, *subsample=1*, *los='z'*, *poles=[]*)

Compute the correlation function by direct pair summation, either as a function of separation (*R*) or as a function of separation and line-of-sight angle (*R*, *mu*)

The estimator used to compute the correlation function is:

$$\xi(r,\mu) = DD(r,\mu)/RR(r,\mu) - 1.$$

where *DD* is the number of data-data pairs, and *RR* is the number of random-random pairs, which is determined solely by the binning used, assuming a constant number density

Parameters **datasources** : list of DataSource objects

the list of data instances from which the 3D correlation will be computed

redges : array_like

the bin edges for the *R* variable

Nmu : int, optional

the number of desired *mu* bins, where *mu* is the cosine of the angle from the line-of-sight. Default is *0*, in which case the correlation function is binned as a function of *R* only

comm : MPI.Communicator, optional

the communicator to pass to the `ParticleMesh` object. If not provided, `MPI.COMM_WORLD` is used

subsample : int, optional

downsample the input datasources by choosing 1 out of every *N* points. Default is *1* (no subsampling).

los : str, {'x', 'y', 'z'}, optional

the dimension to treat as the line-of-sight; default is 'z'.

poles : list of int, optional

integers specifying the multipoles to compute from the 2D correlation function

Returns **pc** : `kdcount.correlate.paircount`

the pair counting instance

xi : array_like

the correlation function result; if *poles* supplied, the shape is *(len(redges)-1, len(poles))*, otherwise, the shape is either *(len(redges)-1,)* or *(len(redges)-1, Nmu)*

RR : array_like

the number of random-random pairs (used as normalization of the data-data pairs) `nbodykit.measurestats.`**`project_to_basis`**(*comm*, *x3d*, *y3d*, *edges*, *los='z'*, *poles=[]*, *hermitian_symmetric=False*)

Project a 3D statistic on to the specified basis. The basis will be one of:

- 2D (*x*, *mu*) bins: *mu* is the cosine of the angle to the line-of-sight
- 2D (*x*, *ell*) bins: *ell* is the multipole number, which specifies the Legendre polynomial when weighting different *mu* bins

Note: The 2D (*x*, *mu*) bins will be computed only if *poles* is specified. See return types for further details.

Parameters **comm** : MPI.Communicatior

the MPI communicator

x3d : list of array_like

list of coordinate values for each dimension of the *y3d* array; the items must able to be broadcasted to the same shape as *y3d*

y3d : array_like, real or complex

the 3D array holding the statistic to be projected to the specified basis

edges : list of arrays, (2,)

list of arrays specifying the edges of the desired *x* bins and *mu* bins

los : str, {'x','y','z'}, optional

the line-of-sight direction to use, which *mu* is defined with respect to; default is *z*.

poles : list of int, optional

if provided, a list of integers specifying multipole numbers to project the 2d *(x, mu)* bins on to

hermitian_symmetric : bool, optional

Whether the input array *y3d* is Hermitian-symmetric, i.e., the negative frequency terms are just the complex conjugates of the corresponding positive-frequency terms; if `True`, the positive frequency terms will be explicitly double-counted to account for this symmetry

Returns **result** : tuple

the 2D binned results; a tuple of `(xmean_2d, mumean_2d, y2d, N_2d)`, where:

xmean_2d [array_like, (Nx, Nmu)] the mean *x* value in each 2D bin

mumean_2d [array_like, (Nx, Nmu)] the mean *mu* value in each 2D bin

y2d [array_like, (Nx, Nmu)] the mean *y3d* value in each 2D bin

N_2d [array_like, (Nx, Nmu)] the number of values averaged in each 2D bin

pole_result : tuple or *None*

the multipole results; if *poles* supplied it is a tuple of `(xmean_1d, poles, N_1d)`, where:

xmean_1d [array_like, (Nx,)] the mean *x* value in each 1D multipole bin

poles [array_like, (Nell, Nx)] the mean multipoles value in each 1D bin

N_1d [array_like, (Nx,)] the number of values averaged in each 1D bin

Notes

- the *mu* range extends from 0.0 to 1.0
- the *mu* bins are half-inclusive half-exclusive, except the last bin is inclusive on both ends (to include *mu = 1.0*)

nbodykit.mockmaker module

`nbodykit.mockmaker.`**`gaussian_complex_fields`**(*pm*, *linear_power*, *compute_displacement=False*)

Make a Gaussian realization of a overdensity field, $\delta(x)$

If specified, also compute the corresponding linear Zel'dovich displacement field $\psi(x)$, which is related to the linear velocity field via:

$$v(x) = \frac{\psi(x)}{faH}$$

Parameters **pm** : pmesh.pm.ParticleMesh

the mesh object

linear_power : callable

a function taking wavenumber as its only argument, which returns the linear power spectrum

compute_displacement : bool, optional

if `True`, also return the linear Zel'dovich displacement field; default is `False`

Returns **delta_k** : ComplexField

the real-space Gaussian overdensity field

disp_k : ComplexField or `None`

if requested, the Gaussian displacement field

Notes

This computes the overdensity field using the following steps:

1. Generate random variates from $\mathcal{N}(\mu = 0, \sigma = 1)$
2. FFT the above field from configuration to Fourier space
3. Scale the Fourier field by $(P(k)N^3/V)^{1/2}$

After step 2, the field has a variance of N^{-3} (using the normalization convention of *pmesh*), since the variance of the complex FFT (with no additional normalization) is $N^3 \times \sigma^2_{\rm real}$ and *pmesh* divides each field by N^3.

Furthermore, the power spectrum is defined as V * variance. Thus, the extra factor of N**3 / V that shows up in step 3, cancels this factor such that the power spectrum is P(k).

The linear displacement field is computed as:

$$\psi_i(k) = i\frac{k_i}{k^2}\delta(k)$$

Note: To recover the linear velocity in proper units, i.e., km/s, from the linear displacement, an additional factor of $f \times a \times H(a)$ is required

---

`nbodykit.mockmaker.`**`gaussian_real_fields`**(*pm*, *linear_power*, *compute_displacement=False*)

Make a Gaussian realization of a overdensity field in real-space $\delta(x)$

If specified, also compute the corresponding linear Zel'dovich displacement field $\psi(x)$, which is related to the linear velocity field via:

Parameters **pm** : pmesh.pm.ParticleMesh

the mesh object

linear_power : callable

a function taking wavenumber as its only argument, which returns the linear power spectrum

compute_displacement : bool, optional

if `True`, also return the linear Zel'dovich displacement field; default is `False`

Returns **delta** : RealField

the real-space Gaussian overdensity field

disp : RealField or `None`

if requested, the Gaussian displacement field

Notes

See the docstring for *`gaussian_complex_fields()`* for the steps involved in generating the fields

`nbodykit.mockmaker.`**`lognormal_transform`**(*density*, *bias=1.0*)

Apply a (biased) lognormal transformation of the density field by computing:

$$F(\delta) = \frac{1}{N}e^{b*\delta}$$

where δ is the initial overdensity field and the normalization N is chosen such that $\langle F(\delta)\rangle = 1$

Parameters **density** : array_like

the input density field to apply the transformation to

bias : float, optional

optionally apply a linear bias to the density field; default is unbiased (1.0)

Returns **toret** : RealField

the real field holding the transformed density field `nbodykit.mockmaker.`**`poisson_sample_to_points`**(*delta*, *displacement*, *pm*, *nbar*, *rsd=None*, *f=0.0*, *bias=1.0*)

Poisson sample the linear delta and displacement fields to points.

This applies the following steps:

1.use a (biased) lognormal transformation to make the overdensity field positive-definite

2.use the Zel'dovich displacement field to mimic nonlinear growth of structure

3.poisson sample the overdensity field to points, disributing particles uniformly within the mesh cells

Parameters **delta** : RealField

the linear overdensity field to sample

displacement : list of RealField (3,)

the linear displacement fields which is used to move the particles

nbar : float

the desired number density of the output catalog of objects

rsd : {'x', 'y' 'z'}

the direction to apply RSD to; if `None` (default), no RSD will be added

f : float, optional

the growth rate, equal to $f = dlnDdlna$, which scales the strength of the RSD; default is 0. (no RSD)

bias : float, optional

apply a linear bias to the overdensity field (default is 1.)

Returns **pos** : array_like, (N, 3)

the Cartesian positions of the particles in the box

nbodykit.ndarray module

Helper routines for ndarray objects

These are really handy things if numpy had them.

`nbodykit.ndarray.`**`equiv_class`**(*labels*, *values*, *op*, *dense_labels=False*, *identity=None*, *minlength=None*)

apply operation to equivalent classes by label, on values

Parameters **labels** : array_like

the label of objects, starting from 0.

values : array_like

the values of objects (len(labels), ...)

op : `numpy.ufunc`

the operation to apply

dense_labels : boolean

If the labels are already dense (from 0 to Nobjects - 1) If False, `numpy.unique()` is used to convert the labels internally

Returns result :

the value of each equivalent class

Examples

```
>>> x = numpy.arange(10)
>>> print equiv_class(x, x, numpy.fmin, dense_labels=True)
[0 1 2 3 4 5 6 7 8 9]
```

```
>>> x = numpy.arange(10)
>>> v = numpy.arange(20).reshape(10, 2)
>>> x[1] = 0
>>> print equiv_class(x, 1.0 * v, numpy.fmin, dense_labels=True, identity=numpy.inf)
[[  0.   1.]
 [ inf  inf]
 [  4.   5.]
 [  6.   7.]
 [  8.   9.]
 [ 10.  11.]
 [ 12.  13.]
 [ 14.  15.]
 [ 16.  17.]
 [ 18.  19.]]
```

`nbodykit.ndarray.`**`extend_dtype`**(*data*, *extra_dtypes*)

Extend the data type of a structured array

Parameters **data** : array_like

a structured array

extra_dtypes : list of (str, dtyple)

a list of data types, specified by the their name and data type

Returns **new** : array_like

a copy of *data*, with the extra data type fields, initialized to zero

`nbodykit.ndarray.`**`replacesorted`**(*arr*, *sorted*, *b*, *out=None*)

replace a with corresponding b in arr

Parameters **arr** : array_like

input array

sorted : array_like

sorted

b : array_like

out : array_like,

output array

Result

——

newarr : array_like

arr with a replaced by corresponding b

Examples

```
>>> print replacesorted(numpy.arange(10), numpy.arange(5), numpy.ones(5))
[1 1 1 1 1 5 6 7 8 9]
```

nbodykit.storage module

class `nbodykit.storage.`**`Measurement1DStorage`**`(`*path*`)`
Bases: *`nbodykit.storage.MeasurementStorage`*

Methods

`create`(dim, path)	
`open`()	
`write`(edges, cols, data, **meta)	Write out a 1D measurement to file

`write``(`*edges, cols, data, **meta*`)`
Write out a 1D measurement to file

Parameters **edges** : array_like

the edges of the bins used for the measurement

cols : list

list of names for the corresponding *data*

data : list of arrays

list of 1D arrays specifying the data, which are written as columns to file; complex arrays will be written as two columns (real and imag).

meta :

Any additional metadata to write to file, specified as keyword arguments

Notes

Any lines not holding data values i.e., *edges* or *metadata*, will start with the # character. This allows the output file to be read using the default arguments of *numpy.loadtxt*, which treats lines beginning with # as comments and skips them.

class `nbodykit.storage.`**`Measurement2DStorage`**`(`*path*`)`
Bases: *`nbodykit.storage.MeasurementStorage`*

Methods

`create`(dim, path)	
`open`()	
`write`(edges, cols, data, **meta)	Write a 2D measurement as plain text.

write (*edges*, *cols*, *data*, ***meta*)

Write a 2D measurement as plain text.

Parameters **edges** : list

list specifying the bin edges in both dimensions

cols : list

list of names for the corresponding *data*

data : list

list of 2D arrays holding the data; complex arrays will be treated as two arrays (real and imag).

meta : dict

any additional metadata to write out at the end of the file

class nbodykit.storage.**MeasurementStorage** (*path*)

Bases: object

Class to write a 1D or 2D measurement to a plaintext file

Methods

create(dim, path)
open()
write(edges, cols, data, **meta)

__init__ (*path*)

classmethod create (*dim*, *path*)

open ()

write (*edges*, *cols*, *data*, ***meta*)

nbodykit.stripedfile module

class nbodykit.stripedfile.**DataStorage** (*path*, *filetype*, *filetype_args={}*)

Bases: object

DataStorage provides a continuous view of across several files.

Methods

create(kls, path, filetype, npart[, ...])	Create a striped file.
get_file(i)	
iter(columns, bunchsize, comm)	Parallel reading.
read(column, start, end)	
write(column, start, data)	this function provides a continuous view of multiple files

`__init__`(*path*, *filetype*, *filetype_args={}*)
Filetype must be of a subclass of StripeFile

classmethod **`create`**(*kls*, *path*, *filetype*, *npart*, *filetype_args={}*)
Create a striped file.

npart is a list of npart for each file

`get_file`(*i*)

`iter`(*columns*, *bunchsize*, *comm*)
Parallel reading. This is a generator function and a collective operation.

Use a *for* loop. For example:

```
for i, P in enumerate(read(comm, 'snapshot', TPMSnapshotFile)):
    ....
    # process P
```

Parameters **comm** : `MPI.Comm`

Communicator

bunchsize : int

Number of particles to read per rank in each iteration. if < 0, all particles are read in one iteration

Yields data in columns

`read`(*column*, *start*, *end*)

`write`(*column*, *start*, *data*)
this function provides a continuous view of multiple files

class `nbodykit.stripedfile.`**`StripeFile`**
Bases: `object`

Methods

`enum`(filetype, basename[, args])	Iterate over all files of the type
`read`(column, mystart, myend)	Read a property column of particles
`readat`(offset, nitem, dtype)	
`write`(column, mystart, data)	
`writeat`(offset, data)	

`column_names` **= {'Position', 'ID', 'Mass', 'Label', 'Velocity'}**

classmethod **`enum`**(*filetype*, *basename*, *args={}*)
Iterate over all files of the type

`read`(*column*, *mystart*, *myend*)
Read a property column of particles

Parameters **mystart** : int

offset to start reading within this file. (inclusive)

myend : int

offset to end reading within this file. (exclusive)

Returns data : array_like (myend - mystart)

data in unspecified units.

readat (*offset*, *nitem*, *dtype*)

write (*column*, *mystart*, *data*)

writeat (*offset*, *data*)

nbodykit.version module

Maintainer's corner

Here, we outline notes intended for the maintainers of nbodykit, includuing details on running the unit test suite and tagging and uploading releases.

Dependency of tests

- pytest
- pytest-pipeline
- pytest-cov
- coveralls

To run the tests use

```
py.test nbodykit
```

or a specific test

```
py.test nbodykit/test/test_batch.py::TestStdin
```

Steps to tag a release

For a stable release

1. Bump the version, in `nbodykit/__init__.py`, removing `.dev0` postfix;
2. Add a git commit, add the version tag, push to master.
3. `python setup.py sdist`
4. Upload the .tar.gz file in `dist/` to pypi.
5. Bump the version, in `nbodykit/__init__.py`, raise the version number, add *.dev0* postfix
6. Add a git commit, push to master.

For a development release

1. Bump the version, in `nbodykit/__init__.py`, increase the release revision number in `.dev0` postfix;
2. Add a git commit, add the version tag, push to master.
3. `python setup.py sdist`
4. Add a git commit, push to master.

We shall have a nightly build on NERSC that builds three bundles in `/usr/common/contrib/bccp/nbodykit`:

- `nbodykit-latest.tar.gz` : built from the HEAD of the master branch
- `nbodykit-stable.tar.gz` : built from the latest version of nbodykit uploaded to pypi, which will only be built if the current bundle is out of date
- `nbodykit-dep.tar.gz` : build from the dependencies in `requirements.txt`, which will only be built if one of the current dependencies is out of date

CHAPTER 2

Get in touch

- Report bugs, suggest feature ideas, or view the source code on GitHub.

n

nbodykit, 92
nbodykit.core, 26
nbodykit.core.algorithms, 32
nbodykit.core.algorithms.BianchiFFTPower, 33
nbodykit.core.algorithms.ExampleAlgorithm, 36
nbodykit.core.algorithms.FiberCollisions, 39
nbodykit.core.algorithms.FOF, 37
nbodykit.core.algorithms.FOF6D, 38
nbodykit.core.algorithms.PaintGrid, 40
nbodykit.core.algorithms.PairCountCorrelation, 42
nbodykit.core.algorithms.PeriodicBox, 44
nbodykit.core.algorithms.RedshiftHistogram, 49
nbodykit.core.algorithms.Subsample, 50
nbodykit.core.algorithms.TestBoxSize, 51
nbodykit.core.algorithms.TidalTensor, 52
nbodykit.core.algorithms.TraceHalo, 53
nbodykit.core.datasource, 54
nbodykit.core.datasource.BigFileGrid, 59
nbodykit.core.datasource.FastPM, 61
nbodykit.core.datasource.FOFGroups, 60
nbodykit.core.datasource.Gadget, 62
nbodykit.core.datasource.HaloLabel, 65
nbodykit.core.datasource.MultiFile, 66
nbodykit.core.datasource.Pandas, 67
nbodykit.core.datasource.PlainText, 68
nbodykit.core.datasource.RaDecRedshift, 70
nbodykit.core.datasource.ShiftedObserver, 72
nbodykit.core.datasource.Subsample, 73
nbodykit.core.datasource.TPMLabel, 74
nbodykit.core.datasource.TPMSnapshot, 75
nbodykit.core.datasource.UniformBox, 76
nbodykit.core.datasource.ZeldovichSim, 77
nbodykit.core.datasource.Zheng07HOD, 78
nbodykit.core.painter, 80
nbodykit.core.painter.DefaultPainter, 82
nbodykit.core.painter.MomentumPainter, 83
nbodykit.core.source, 84
nbodykit.core.source.Grid, 86
nbodykit.core.source.Particle, 87
nbodykit.core.transfer, 88
nbodykit.core.transfer.TransferFunction, 89
nbodykit.cosmology, 125
nbodykit.dataset, 126
nbodykit.distributedarray, 135
nbodykit.files, 137
nbodykit.fkp, 140
nbodykit.fof, 143
nbodykit.halos, 144
nbodykit.io, 99
nbodykit.io.bigfile, 101
nbodykit.io.binary, 102
nbodykit.io.csv, 104
nbodykit.io.stack, 105
nbodykit.io.tools, 106
nbodykit.io.tpm, 108
nbodykit.measurestats, 145
nbodykit.mockmaker, 149
nbodykit.ndarray, 151
nbodykit.plugins, 108
nbodykit.plugins.fromfile, 110
nbodykit.plugins.hooks, 112
nbodykit.plugins.manager, 112
nbodykit.plugins.schema, 114
nbodykit.plugins.validate, 117

nbodykit.storage, 153
nbodykit.stripedfile, 154
nbodykit.utils, 117
nbodykit.utils.meshtools, 118
nbodykit.utils.selectionlanguage, 120
nbodykit.utils.taskmanager, 124
nbodykit.version, 156

Symbols

__init__() (nbodykit.core.Painter method), 31, 98
__init__() (nbodykit.core.algorithms.BianchiFFTPower.BianchiPowerAlgorithm method), 34
__init__() (nbodykit.core.algorithms.ExampleAlgorithm.Describe method), 36
__init__() (nbodykit.core.algorithms.ExampleAlgorithm.Play method), 37
__init__() (nbodykit.core.algorithms.FOF.FOFAlgorithm method), 37
__init__() (nbodykit.core.algorithms.FOF6D.FOF6DAlgorithm method), 38
__init__() (nbodykit.core.algorithms.FiberCollisions.FiberCollisionsAlgorithm method), 40
__init__() (nbodykit.core.algorithms.PaintGrid.PaintGridAlgorithm method), 41
__init__() (nbodykit.core.algorithms.PairCountCorrelation.PairCountCorrelationAlgorithm method), 43
__init__() (nbodykit.core.algorithms.PeriodicBox.FFTCorrelationAlgorithm method), 45
__init__() (nbodykit.core.algorithms.PeriodicBox.FFTPowerAlgorithm method), 47
__init__() (nbodykit.core.algorithms.RedshiftHistogram.RedshiftHistogramAlgorithm method), 49
__init__() (nbodykit.core.algorithms.Subsample.Subsample method), 51
__init__() (nbodykit.core.algorithms.TestBoxSize.TestBoxSizeAlgorithm method), 52
__init__() (nbodykit.core.algorithms.TidalTensor.TidalTensor method), 53
__init__() (nbodykit.core.algorithms.TraceHalo.TraceHaloAlgorithm method), 54
__init__() (nbodykit.core.datasource.BigFileGrid.BigFileGridSource method), 59
__init__() (nbodykit.core.datasource.DataCache method), 54
__init__() (nbodykit.core.datasource.DataStream method), 58
__init__() (nbodykit.core.datasource.FOFGroups.FOFDataSource method), 60
__init__() (nbodykit.core.datasource.FastPM.FastPMDataSource method), 62
__init__() (nbodykit.core.datasource.Gadget.GadgetDataSource method), 63
__init__() (nbodykit.core.datasource.Gadget.GadgetGroupTabDataSource method), 64
__init__() (nbodykit.core.datasource.HaloLabel.HaloLabel method), 65
__init__() (nbodykit.core.datasource.MultiFile.MultiFileDataSource method), 66
__init__() (nbodykit.core.datasource.Pandas.PandasDataSource method), 67
__init__() (nbodykit.core.datasource.PlainText.PlainTextDataSource method), 69
__init__() (nbodykit.core.datasource.RaDecRedshift.RaDecRedshiftDataSo method), 71
__init__() (nbodykit.core.datasource.ShiftedObserver.ShiftedObserverDataS method), 72
__init__() (nbodykit.core.datasource.Subsample.SubsampleDataSource method), 73
__init__() (nbodykit.core.datasource.TPMLabel.TPMLabel method), 74
__init__() (nbodykit.core.datasource.TPMSnapshot.TPMSnapshotDataSour method), 75
__init__() (nbodykit.core.datasource.UniformBox.UniformBoxDataSource method), 76
__init__() (nbodykit.core.datasource.ZeldovichSim.ZeldovichSimDataSourc method), 78
__init__() (nbodykit.core.datasource.Zheng07HOD.Zheng07HodDataSourc method), 79
__init__() (nbodykit.core.painter.DefaultPainter.DefaultPainter method), 82
__init__() (nbodykit.core.painter.MomentumPainter.MomentumPainter method), 83
__init__() (nbodykit.core.painter.Painter method), 81
__init__() (nbodykit.core.source.Grid.GridSource method), 86
__init__() (nbodykit.core.source.Painter method), 84
__init__() (nbodykit.core.source.Particle.ParticleSource method), 87
__init__() (nbodykit.core.transfer.TransferFunction.AnisotropicCIC

method), 89
__init__() (nbodykit.core.transfer.TransferFunction.AnisotropicTSC method), 90
__init__() (nbodykit.core.transfer.TransferFunction.CICWindow method), 90
__init__() (nbodykit.core.transfer.TransferFunction.NormalizeDC method), 91
__init__() (nbodykit.core.transfer.TransferFunction.RemoveDC method), 91
__init__() (nbodykit.core.transfer.TransferFunction.TSCWindow method), 92
__init__() (nbodykit.cosmology.Cosmology method), 125
__init__() (nbodykit.cosmology.CosmologyBase.sampled_function method), 126
__init__() (nbodykit.dataset.Corr1dDataSet method), 127
__init__() (nbodykit.dataset.Corr2dDataSet method), 128
__init__() (nbodykit.dataset.DataSet method), 129
__init__() (nbodykit.dataset.Power1dDataSet method), 133
__init__() (nbodykit.dataset.Power2dDataSet method), 134
__init__() (nbodykit.distributedarray.DistributedArray method), 135
__init__() (nbodykit.distributedarray.LinearTopology method), 136
__init__() (nbodykit.files.GadgetGroupTabFile method), 137
__init__() (nbodykit.files.GadgetSnapshotFile method), 138
__init__() (nbodykit.files.HaloLabelFile method), 138
__init__() (nbodykit.files.Snapshot method), 139
__init__() (nbodykit.files.TPMSnapshotFile method), 140
__init__() (nbodykit.fkp.FKPCatalog method), 140
__init__() (nbodykit.io.bigfile.BigFile method), 102
__init__() (nbodykit.io.binary.BinaryFile method), 103
__init__() (nbodykit.io.csv.CSVFile method), 104
__init__() (nbodykit.io.stack.FileStack method), 106
__init__() (nbodykit.io.tpm.TPMBinaryFile method), 108
__init__() (nbodykit.plugins.PluginBase method), 109
__init__() (nbodykit.plugins.manager.PluginManager method), 112
__init__() (nbodykit.plugins.schema.ConstructorSchema method), 115
__init__() (nbodykit.storage.MeasurementStorage method), 154
__init__() (nbodykit.stripedfile.DataStorage method), 155
__init__() (nbodykit.utils.meshtools.MeshSlab method), 118
__init__() (nbodykit.utils.selectionlanguage.BoolBinOp method), 120
__init__() (nbodykit.utils.selectionlanguage.BoolNot method), 120
__init__() (nbodykit.utils.selectionlanguage.ColumnSlice method), 121
__init__() (nbodykit.utils.selectionlanguage.CompOperand method), 121
__init__() (nbodykit.utils.selectionlanguage.CompOperator method), 122
__init__() (nbodykit.utils.selectionlanguage.LeftCompOperand method), 122
__init__() (nbodykit.utils.selectionlanguage.Query method), 123
__init__() (nbodykit.utils.selectionlanguage.RightCompOperand method), 123
__init__() (nbodykit.utils.taskmanager.TaskManager method), 124

A

add_and_validate_schema() (in module nbodykit.plugins.hooks), 112
add_argument() (nbodykit.plugins.schema.ConstructorSchema method), 115
add_logger() (in module nbodykit.plugins.hooks), 112
add_user_plugin() (nbodykit.plugins.manager.PluginManager method), 112
Algorithm (class in nbodykit.core), 26, 93
Algorithm (class in nbodykit.core.algorithms), 32
alpha (nbodykit.fkp.FKPCatalog attribute), 141
AnisotropicCIC (class in nbodykit.core.transfer.TransferFunction), 89
AnisotropicTSC (class in nbodykit.core.transfer.TransferFunction), 89
apply_bianchi_kernel() (in module nbodykit.measurestats), 145
Argument (class in nbodykit.plugins.schema), 114
ArgumentBase (in module nbodykit.plugins.schema), 114
asarray() (nbodykit.io.FileType method), 100
attach_comm() (in module nbodykit.plugins.hooks), 112
attach_cosmo() (in module nbodykit.plugins.hooks), 112
attribute() (in module nbodykit.plugins.schema), 116
attrs (nbodykit.core.Source attribute), 27, 94
attrs (nbodykit.core.source.Grid.GridSource attribute), 87
attrs (nbodykit.core.source.Particle.ParticleSource attribute), 88
attrs (nbodykit.core.source.Source attribute), 85
attrs (nbodykit.io.stack.FileStack attribute), 106
average() (nbodykit.dataset.DataSet method), 130

B

basepaint() (nbodykit.core.Painter method), 31, 98
basepaint() (nbodykit.core.painter.Painter method), 81
BianchiPowerAlgorithm (class in nbodykit.core.algorithms.BianchiFFTPower),

33
BigFile (class in nbodykit.io.bigfile), 101
BigFileGridSource (class in nbodykit.core.datasource.BigFileGrid), 59
bin_ndarray() (in module nbodykit.dataset), 134
BinaryFile (class in nbodykit.io.binary), 102
bincount() (nbodykit.distributedarray.DistributedArray method), 135
binning_type() (in module nbodykit.core.algorithms.PairCountCorrelation), 44
BoolAnd (class in nbodykit.utils.selectionlanguage), 120
BoolBinOp (class in nbodykit.utils.selectionlanguage), 120
BoolNot (class in nbodykit.utils.selectionlanguage), 120
BoolOr (class in nbodykit.utils.selectionlanguage), 120
BoxSize (nbodykit.core.Source attribute), 27, 94
BoxSize (nbodykit.core.source.Source attribute), 85
BoxSizeParser() (nbodykit.core.datasource.DataSourceBase static method), 57

C

case_insensitive_name_match() (in module nbodykit.plugins.fromfile), 111
cast() (nbodykit.plugins.schema.ConstructorSchema static method), 116
centerofmass() (in module nbodykit.halos), 144
CICWindow (class in nbodykit.core.transfer.TransferFunction), 90
close() (nbodykit.core.datasource.DataStream method), 58
close() (nbodykit.fkp.FKPCatalog method), 141
closed (nbodykit.core.datasource.DataStream attribute), 58
closed (nbodykit.fkp.FKPCatalog attribute), 141
column_names (nbodykit.stripedfile.StripeFile attribute), 155
columns (nbodykit.core.Source attribute), 27, 94
columns (nbodykit.core.source.Grid.GridSource attribute), 87
columns (nbodykit.core.source.Particle.ParticleSource attribute), 88
columns (nbodykit.core.source.Source attribute), 85
columns (nbodykit.io.FileType attribute), 100
ColumnSlice (class in nbodykit.utils.selectionlanguage), 121
comoving_distance() (nbodykit.cosmology.Cosmology method), 125
CompOperand (class in nbodykit.utils.selectionlanguage), 121
CompOperator (class in nbodykit.utils.selectionlanguage), 121
compute() (nbodykit.core.Source static method), 27, 94
compute() (nbodykit.core.source.Source static method), 85
compute() (nbodykit.utils.taskmanager.TaskManager method), 124
compute_3d_corr() (in module nbodykit.measurestats), 145
compute_3d_power() (in module nbodykit.measurestats), 145
compute_bianchi_poles() (in module nbodykit.measurestats), 146
compute_brutal_corr() (in module nbodykit.measurestats), 147
concat() (nbodykit.utils.selectionlanguage.RightCompOperand method), 123
ConfigurationError, 110
ConstructorSchema (class in nbodykit.plugins.schema), 114
ConstructorSchema.Argument (class in nbodykit.plugins.schema), 115
contains() (nbodykit.plugins.schema.ConstructorSchema method), 116
coords() (nbodykit.utils.meshtools.MeshSlab method), 118
copy() (nbodykit.dataset.DataSet method), 130
Corr1dDataSet (class in nbodykit.dataset), 126
Corr2dDataSet (class in nbodykit.dataset), 127
Cosmology (class in nbodykit.cosmology), 125
CosmologyBase (class in nbodykit.cosmology), 126
CosmologyBase.sampled_function (class in nbodykit.cosmology), 126
count() (in module nbodykit.halos), 145
create() (nbodykit.files.Snapshot class method), 139
create() (nbodykit.files.TPMSnapshotFile class method), 140
create() (nbodykit.plugins.manager.PluginManager class method), 113
create() (nbodykit.plugins.PluginBase class method), 109
create() (nbodykit.storage.MeasurementStorage class method), 154
create() (nbodykit.stripedfile.DataStorage class method), 155
csv_partition_sizes() (in module nbodykit.io.tools), 106
CSVFile (class in nbodykit.io.csv), 104

D

data (nbodykit.fkp.FKPCatalog attribute), 141
DataCache (class in nbodykit.core.datasource), 54
DataSet (class in nbodykit.dataset), 128
DataSource (class in nbodykit.core), 27, 95
DataSource (class in nbodykit.core.datasource), 55
DataSource.MissingColumn, 28, 56, 95
DataSourceBase (class in nbodykit.core.datasource), 57
DataStorage (class in nbodykit.stripedfile), 154
DataStream (class in nbodykit.core.datasource), 57

default_columns (nbodykit.fkp.FKPCatalog attribute), 141
DefaultPainter (class in nbodykit.core.painter.DefaultPainter), 82
Describe (class in nbodykit.core.algorithms.ExampleAlgorithm), 36
DistributedArray (class in nbodykit.distributedarray), 135

E

empty (nbodykit.core.datasource.DataCache attribute), 55
EmptyConfigurationError, 110
EmptyRankType (class in nbodykit.distributedarray), 136
enum() (in module nbodykit.utils.taskmanager), 125
enum() (nbodykit.stripedfile.StripeFile class method), 155
equiv_class() (in module nbodykit.ndarray), 151
eval() (nbodykit.utils.selectionlanguage.BoolAnd method), 120
eval() (nbodykit.utils.selectionlanguage.BoolBinOp method), 120
eval() (nbodykit.utils.selectionlanguage.BoolNot method), 120
eval() (nbodykit.utils.selectionlanguage.BoolOr method), 121
eval() (nbodykit.utils.selectionlanguage.CompOperand method), 121
eval() (nbodykit.utils.selectionlanguage.CompOperator method), 122
eval() (nbodykit.utils.selectionlanguage.LeftCompOperand method), 122
eval() (nbodykit.utils.selectionlanguage.RightCompOperand method), 123
extend_dtype() (in module nbodykit.ndarray), 152

F

FastPMDataSource (class in nbodykit.core.datasource.FastPM), 61
FFTCorrelationAlgorithm (class in nbodykit.core.algorithms.PeriodicBox), 44
FFTPowerAlgorithm (class in nbodykit.core.algorithms.PeriodicBox), 46
FiberCollisionsAlgorithm (class in nbodykit.core.algorithms.FiberCollisions), 39
FieldType() (in module nbodykit.core.algorithms.PeriodicBox), 48
FileStack (class in nbodykit.io.stack), 105
FileType (class in nbodykit.io), 99
fill_namespace() (in module nbodykit.plugins.fromfile), 111
fill_schema() (nbodykit.core.algorithms.BianchiFFTPower.BianchiPowerAlgorithm class method), 35
fill_schema() (nbodykit.core.algorithms.ExampleAlgorithm.Describe class method), 36
fill_schema() (nbodykit.core.algorithms.ExampleAlgorithm.Play class method), 37
fill_schema() (nbodykit.core.algorithms.FiberCollisions.FiberCollisionsAlg class method), 40
fill_schema() (nbodykit.core.algorithms.FOF.FOFAlgorithm class method), 38
fill_schema() (nbodykit.core.algorithms.FOF6D.FOF6DAlgorithm class method), 39
fill_schema() (nbodykit.core.algorithms.PaintGrid.PaintGridAlgorithm class method), 41
fill_schema() (nbodykit.core.algorithms.PairCountCorrelation.PairCountCo class method), 43
fill_schema() (nbodykit.core.algorithms.PeriodicBox.FFTCorrelationAlgori class method), 46
fill_schema() (nbodykit.core.algorithms.PeriodicBox.FFTPowerAlgorithm class method), 48
fill_schema() (nbodykit.core.algorithms.RedshiftHistogram.RedshiftHistogr class method), 50
fill_schema() (nbodykit.core.algorithms.Subsample.Subsample class method), 51
fill_schema() (nbodykit.core.algorithms.TestBoxSize.TestBoxSizeAlgorithm class method), 52
fill_schema() (nbodykit.core.algorithms.TidalTensor.TidalTensor class method), 53
fill_schema() (nbodykit.core.algorithms.TraceHalo.TraceHaloAlgorithm class method), 54
fill_schema() (nbodykit.core.datasource.BigFileGrid.BigFileGridSource class method), 60
fill_schema() (nbodykit.core.datasource.FastPM.FastPMDataSource class method), 62
fill_schema() (nbodykit.core.datasource.FOFGroups.FOFDataSource class method), 61
fill_schema() (nbodykit.core.datasource.Gadget.GadgetDataSource class method), 63
fill_schema() (nbodykit.core.datasource.Gadget.GadgetGroupTabDataSourc class method), 65
fill_schema() (nbodykit.core.datasource.HaloLabel.HaloLabel class method), 65
fill_schema() (nbodykit.core.datasource.MultiFile.MultiFileDataSource class method), 66
fill_schema() (nbodykit.core.datasource.Pandas.PandasDataSource class method), 68
fill_schema() (nbodykit.core.datasource.PlainText.PlainTextDataSource class method), 70
fill_schema() (nbodykit.core.datasource.RaDecRedshift.RaDecRedshiftData class method), 71
fill_schema() (nbodykit.core.datasource.ShiftedObserver.ShiftedObserverDa class method), 72
fill_schema() (nbodykit.core.datasource.Subsample.SubsampleDataSource class method), 74
fill_schema() (nbodykit.core.datasource.TPMLabel.TPMLabel class method), 74

fill_schema() (nbodykit.core.datasource.TPMSnapshot.TPMSnapshotDataSource class method), 75
fill_schema() (nbodykit.core.datasource.UniformBox.UniformBoxDataSource class method), 77
fill_schema() (nbodykit.core.datasource.ZeldovichSim.ZeldovichSimDataSource class method), 78
fill_schema() (nbodykit.core.datasource.Zheng07HOD.Zheng07HodDataSource class method), 80
fill_schema() (nbodykit.core.painter.DefaultPainter.DefaultPainter class method), 82
fill_schema() (nbodykit.core.painter.MomentumPainter.MomentumPainter class method), 84
fill_schema() (nbodykit.core.source.Grid.GridSource class method), 87
fill_schema() (nbodykit.core.source.Particle.ParticleSource class method), 88
fill_schema() (nbodykit.core.transfer.TransferFunction.AnisotropicCIC class method), 89
fill_schema() (nbodykit.core.transfer.TransferFunction.AnisotropicTSC class method), 90
fill_schema() (nbodykit.core.transfer.TransferFunction.CICWindow class method), 90
fill_schema() (nbodykit.core.transfer.TransferFunction.NormalizeDC class method), 91
fill_schema() (nbodykit.core.transfer.TransferFunction.RemoveDC class method), 91
fill_schema() (nbodykit.core.transfer.TransferFunction.TSCWindow class method), 92
fill_schema() (nbodykit.io.bigfile.BigFile class method), 102
fill_schema() (nbodykit.io.binary.BinaryFile class method), 103
fill_schema() (nbodykit.io.csv.CSVFile class method), 105
fill_schema() (nbodykit.io.stack.FileStack class method), 106
fill_schema() (nbodykit.io.tpm.TPMBinaryFile class method), 108
fill_schema() (nbodykit.plugins.PluginBase class method), 110
FKPCatalog (class in nbodykit.fkp), 140
fof() (in module nbodykit.fof), 143
FOF6DAlgorithm (class in nbodykit.core.algorithms.FOF6D), 38
fof_catalogue() (in module nbodykit.fof), 143
fof_halo_label() (in module nbodykit.fof), 144
fof_merge() (in module nbodykit.fof), 144
FOFAlgorithm (class in nbodykit.core.algorithms.FOF), 37
FOFDataSource (class in nbodykit.core.datasource.FOFGroups), 60
format_help() (nbodykit.plugins.manager.PluginManager method), 113
format_help() (nbodykit.plugins.schema.ConstructorSchema

method), 116
from_1d_measurement() (in module nbodykit.dataset), 134
from_2d_measurement() (in module nbodykit.dataset), 134
from_config() (nbodykit.core.source.Painter class method), 84
from_config() (nbodykit.plugins.PluginBase class method), 110
from_files() (nbodykit.io.stack.FileStack class method), 106
from_nbkit() (nbodykit.dataset.Corr1dDataSet class method), 127
from_nbkit() (nbodykit.dataset.Corr2dDataSet class method), 128
from_nbkit() (nbodykit.dataset.DataSet class method), 130
from_nbkit() (nbodykit.dataset.Power1dDataSet class method), 133
from_nbkit() (nbodykit.dataset.Power2dDataSet class method), 134
fsky (nbodykit.fkp.FKPCatalog attribute), 141

G

GadgetDataSource (class in nbodykit.core.datasource.Gadget), 62
GadgetGroupTabDataSource (class in nbodykit.core.datasource.Gadget), 64
GadgetGroupTabFile (class in nbodykit.files), 137
GadgetSnapshotFile (class in nbodykit.files), 137
GatherArray() (in module nbodykit.distributedarray), 136
gaussian_complex_fields() (in module nbodykit.mockmaker), 149
gaussian_real_fields() (in module nbodykit.mockmaker), 150
get() (nbodykit.GlobalComm class method), 92
get() (nbodykit.GlobalCosmology class method), 93
get() (nbodykit.plugins.manager.PluginManager class method), 113
get_dask() (nbodykit.io.FileType method), 100
get_file() (nbodykit.files.Snapshot method), 139
get_file() (nbodykit.stripedfile.DataStorage method), 155
get_file_slice() (in module nbodykit.io.tools), 106
get_init_errmsg() (in module nbodykit.plugins.validate), 117
get_mask() (nbodykit.utils.selectionlanguage.Query method), 123
get_plugin() (nbodykit.plugins.manager.PluginManager method), 113
get_slice_size() (in module nbodykit.io.tools), 107
getsize() (in module nbodykit.io.binary), 103
global_to_local_slice() (in module nbodykit.io.tools), 107
GlobalComm (class in nbodykit), 92
GlobalCosmology (class in nbodykit), 92

GridSource (class in nbodykit.core), 30, 97
GridSource (class in nbodykit.core.datasource), 58
GridSource (class in nbodykit.core.source.Grid), 86

H

HaloLabel (class in nbodykit.core.datasource.HaloLabel), 65
HaloLabelFile (class in nbodykit.files), 138
heads() (nbodykit.distributedarray.LinearTopology method), 136
hermitian_symmetric (nbodykit.utils.meshtools.MeshSlab attribute), 119
hermitian_weights (nbodykit.utils.meshtools.MeshSlab attribute), 119

I

index (nbodykit.utils.meshtools.MeshSlab attribute), 119
infer_csv_dtype() (in module nbodykit.io.tools), 107
interpMake() (in module nbodykit.core.datasource.FastPM), 62
io_extension_points() (in module nbodykit.io), 101
is_float() (in module nbodykit.fkp), 143
is_master() (nbodykit.utils.taskmanager.TaskManager method), 124
is_sampled() (nbodykit.cosmology.CosmologyBase method), 126
is_worker() (nbodykit.utils.taskmanager.TaskManager method), 124
isdefault() (nbodykit.core.datasource.DataStream method), 58
isempty() (nbodykit.utils.selectionlanguage.ColumnSlice method), 121
iter() (nbodykit.stripedfile.DataStorage method), 155

K

keep_cache() (nbodykit.core.DataSource method), 28, 95
keep_cache() (nbodykit.core.datasource.DataSource method), 56
keys() (nbodykit.io.FileType method), 100

L

LeftCompOperand (class in nbodykit.utils.selectionlanguage), 122
LinearTopology (class in nbodykit.distributedarray), 136
ListPluginsAction() (in module nbodykit.plugins), 108
local_fof() (in module nbodykit.fof), 144
local_random_seed() (in module nbodykit.utils), 117
log_populated_stats() (nbodykit.core.datasource.Zheng07HOD.Zheng07HodDataSource method), 80
logger (nbodykit.core.Algorithm attribute), 26, 93
logger (nbodykit.core.algorithms.Algorithm attribute), 32
logger (nbodykit.core.algorithms.BianchiFFTPower.BianchiPowerAlgorithm attribute), 35
logger (nbodykit.core.algorithms.ExampleAlgorithm.Describe attribute), 36
logger (nbodykit.core.algorithms.ExampleAlgorithm.Play attribute), 37
logger (nbodykit.core.algorithms.FiberCollisions.FiberCollisionsAlgorithm attribute), 40
logger (nbodykit.core.algorithms.FOF.FOFAlgorithm attribute), 38
logger (nbodykit.core.algorithms.FOF6D.FOF6DAlgorithm attribute), 39
logger (nbodykit.core.algorithms.PaintGrid.PaintGridAlgorithm attribute), 41
logger (nbodykit.core.algorithms.PairCountCorrelation.PairCountCorrelatio attribute), 43
logger (nbodykit.core.algorithms.PeriodicBox.FFTCorrelationAlgorithm attribute), 46
logger (nbodykit.core.algorithms.PeriodicBox.FFTPowerAlgorithm attribute), 48
logger (nbodykit.core.algorithms.RedshiftHistogram.RedshiftHistogramAlg attribute), 50
logger (nbodykit.core.algorithms.Subsample.Subsample attribute), 51
logger (nbodykit.core.algorithms.TestBoxSize.TestBoxSizeAlgorithm attribute), 52
logger (nbodykit.core.algorithms.TidalTensor.TidalTensor attribute), 53
logger (nbodykit.core.algorithms.TraceHalo.TraceHaloAlgorithm attribute), 54
logger (nbodykit.core.DataSource attribute), 28, 96
logger (nbodykit.core.datasource.BigFileGrid.BigFileGridSource attribute), 60
logger (nbodykit.core.datasource.DataSource attribute), 56
logger (nbodykit.core.datasource.DataSourceBase attribute), 57
logger (nbodykit.core.datasource.FastPM.FastPMDataSource attribute), 62
logger (nbodykit.core.datasource.FOFGroups.FOFDataSource attribute), 61
logger (nbodykit.core.datasource.Gadget.GadgetDataSource attribute), 63
logger (nbodykit.core.datasource.Gadget.GadgetGroupTabDataSource attribute), 65
logger (nbodykit.core.datasource.GridSource attribute), 59
logger (nbodykit.core.datasource.HaloLabel.HaloLabel attribute), 65
logger (nbodykit.core.datasource.MultiFile.MultiFileDataSource attribute), 66
logger (nbodykit.core.datasource.Pandas.PandasDataSource attribute), 68
logger (nbodykit.core.datasource.PlainText.PlainTextDataSource attribute), 70
logger (nbodykit.core.datasource.RaDecRedshift.RaDecRedshiftDataSourc

attribute), 71
logger (nbodykit.core.datasource.ShiftedObserver.ShiftedObserverDataSource attribute), 73
logger (nbodykit.core.datasource.Subsample.SubsampleDataSource attribute), 74
logger (nbodykit.core.datasource.TPMLabel.TPMLabel attribute), 75
logger (nbodykit.core.datasource.TPMSnapshot.TPMSnapshotDataSource attribute), 76
logger (nbodykit.core.datasource.UniformBox.UniformBoxDataSource attribute), 77
logger (nbodykit.core.datasource.ZeldovichSim.ZeldovichSimDataSource attribute), 78
logger (nbodykit.core.datasource.Zheng07HOD.Zheng07HodDataSource attribute), 80
logger (nbodykit.core.GridSource attribute), 30, 97
logger (nbodykit.core.Painter attribute), 31, 98
logger (nbodykit.core.painter.DefaultPainter.DefaultPainter attribute), 83
logger (nbodykit.core.painter.MomentumPainter.MomentumPainter attribute), 84
logger (nbodykit.core.painter.Painter attribute), 81
logger (nbodykit.core.Source attribute), 27, 94
logger (nbodykit.core.source.Grid.GridSource attribute), 87
logger (nbodykit.core.source.Particle.ParticleSource attribute), 88
logger (nbodykit.core.source.Source attribute), 85
logger (nbodykit.core.Transfer attribute), 32, 99
logger (nbodykit.core.transfer.Transfer attribute), 88
logger (nbodykit.core.transfer.TransferFunction.AnisotropicCIC attribute), 89
logger (nbodykit.core.transfer.TransferFunction.AnisotropicTSC attribute), 90
logger (nbodykit.core.transfer.TransferFunction.CICWindow attribute), 90
logger (nbodykit.core.transfer.TransferFunction.NormalizeDC attribute), 91
logger (nbodykit.core.transfer.TransferFunction.RemoveDC attribute), 91
logger (nbodykit.core.transfer.TransferFunction.TSCWindow attribute), 92
logger (nbodykit.io.bigfile.BigFile attribute), 102
logger (nbodykit.io.binary.BinaryFile attribute), 103
logger (nbodykit.io.csv.CSVFile attribute), 105
logger (nbodykit.io.FileType attribute), 100
logger (nbodykit.io.stack.FileStack attribute), 106
logger (nbodykit.io.tpm.TPMBinaryFile attribute), 108
logger (nbodykit.plugins.PluginBase attribute), 110
logger (nbodykit.utils.taskmanager.TaskManager attribute), 125
lognormal_transform() (in module nbodykit.mockmaker), 150

M

Measurement1DStorage (class in nbodykit.storage), 153
Measurement2DStorage (class in nbodykit.storage), 153
MeasurementStorage (class in nbodykit.storage), 154
meshshape (nbodykit.utils.meshtools.MeshSlab attribute), 119
MeshSlab (class in nbodykit.utils.meshtools), 118
MetaclassWithHooks() (in module nbodykit.plugins), 109
model_with_random_seed() (in module nbodykit.core.datasource.Zheng07HOD), 80
MomentumPainter (class in nbodykit.core.painter.MomentumPainter), 83
mu() (nbodykit.utils.meshtools.MeshSlab method), 119
MultiFileDataSource (class in nbodykit.core.datasource.MultiFile), 66

N

nbar (nbodykit.fkp.FKPCatalog attribute), 141
nbodykit (module), 92
nbodykit.core (module), 26, 93
nbodykit.core.algorithms (module), 32
nbodykit.core.algorithms.BianchiFFTPower (module), 33
nbodykit.core.algorithms.ExampleAlgorithm (module), 36
nbodykit.core.algorithms.FiberCollisions (module), 39
nbodykit.core.algorithms.FOF (module), 37
nbodykit.core.algorithms.FOF6D (module), 38
nbodykit.core.algorithms.PaintGrid (module), 40
nbodykit.core.algorithms.PairCountCorrelation (module), 42
nbodykit.core.algorithms.PeriodicBox (module), 44
nbodykit.core.algorithms.RedshiftHistogram (module), 49
nbodykit.core.algorithms.Subsample (module), 50
nbodykit.core.algorithms.TestBoxSize (module), 51
nbodykit.core.algorithms.TidalTensor (module), 52
nbodykit.core.algorithms.TraceHalo (module), 53
nbodykit.core.datasource (module), 54
nbodykit.core.datasource.BigFileGrid (module), 59
nbodykit.core.datasource.FastPM (module), 61
nbodykit.core.datasource.FOFGroups (module), 60
nbodykit.core.datasource.Gadget (module), 62
nbodykit.core.datasource.HaloLabel (module), 65
nbodykit.core.datasource.MultiFile (module), 66
nbodykit.core.datasource.Pandas (module), 67
nbodykit.core.datasource.PlainText (module), 68
nbodykit.core.datasource.RaDecRedshift (module), 70
nbodykit.core.datasource.ShiftedObserver (module), 72
nbodykit.core.datasource.Subsample (module), 73
nbodykit.core.datasource.TPMLabel (module), 74
nbodykit.core.datasource.TPMSnapshot (module), 75
nbodykit.core.datasource.UniformBox (module), 76

nbodykit.core.datasource.ZeldovichSim (module), 77
nbodykit.core.datasource.Zheng07HOD (module), 78
nbodykit.core.painter (module), 80
nbodykit.core.painter.DefaultPainter (module), 82
nbodykit.core.painter.MomentumPainter (module), 83
nbodykit.core.source (module), 84
nbodykit.core.source.Grid (module), 86
nbodykit.core.source.Particle (module), 87
nbodykit.core.transfer (module), 88
nbodykit.core.transfer.TransferFunction (module), 89
nbodykit.cosmology (module), 125
nbodykit.dataset (module), 126
nbodykit.distributedarray (module), 135
nbodykit.files (module), 137
nbodykit.fkp (module), 140
nbodykit.fof (module), 143
nbodykit.halos (module), 144
nbodykit.io (module), 99
nbodykit.io.bigfile (module), 101
nbodykit.io.binary (module), 102
nbodykit.io.csv (module), 104
nbodykit.io.stack (module), 105
nbodykit.io.tools (module), 106
nbodykit.io.tpm (module), 108
nbodykit.measurestats (module), 145
nbodykit.mockmaker (module), 149
nbodykit.ndarray (module), 151
nbodykit.plugins (module), 108
nbodykit.plugins.fromfile (module), 110
nbodykit.plugins.hooks (module), 112
nbodykit.plugins.manager (module), 112
nbodykit.plugins.schema (module), 114
nbodykit.plugins.validate (module), 117
nbodykit.storage (module), 153
nbodykit.stripedfile (module), 154
nbodykit.utils (module), 117
nbodykit.utils.meshtools (module), 118
nbodykit.utils.selectionlanguage (module), 120
nbodykit.utils.taskmanager (module), 124
nbodykit.version (module), 156
ncol (nbodykit.io.FileType attribute), 100
neutrino_mass() (in module nbodykit.cosmology), 126
next() (nbodykit.distributedarray.LinearTopology method), 136
nfiles (nbodykit.io.stack.FileStack attribute), 106
nonsingular (nbodykit.utils.meshtools.MeshSlab attribute), 119
norm2() (nbodykit.utils.meshtools.MeshSlab method), 119
NormalizeDC (class in nbodykit.core.transfer.TransferFunction), 90
NormalizeDC() (nbodykit.core.algorithms.TidalTensor.TidalTensor method), 53

O

open() (nbodykit.core.DataSource method), 28, 96
open() (nbodykit.core.datasource.DataSource method), 56
open() (nbodykit.core.datasource.ShiftedObserver.ShiftedObserverDataSou
method), 73
open() (nbodykit.fkp.FKPCatalog method), 142
open() (nbodykit.storage.MeasurementStorage method), 154
ops (nbodykit.utils.selectionlanguage.CompOperator attribute), 122

P

paint() (nbodykit.core.Painter method), 31, 98
paint() (nbodykit.core.painter.DefaultPainter.DefaultPainter method), 83
paint() (nbodykit.core.painter.MomentumPainter.MomentumPainter method), 84
paint() (nbodykit.core.painter.Painter method), 81
paint() (nbodykit.core.Source method), 27, 94
paint() (nbodykit.core.source.Grid.GridSource method), 87
paint() (nbodykit.core.source.Painter method), 84
paint() (nbodykit.core.source.Particle.ParticleSource method), 88
paint() (nbodykit.core.source.Source method), 85
paint() (nbodykit.fkp.FKPCatalog method), 142
Painter (class in nbodykit.core), 30, 97
Painter (class in nbodykit.core.painter), 80
Painter (class in nbodykit.core.source), 84
PaintGridAlgorithm (class in nbodykit.core.algorithms.PaintGrid), 40
PairCountCorrelationAlgorithm (class in nbodykit.core.algorithms.PairCountCorrelation), 42
PandasDataSource (class in nbodykit.core.datasource.Pandas), 67
parallel_read() (nbodykit.core.DataSource method), 29, 96
parallel_read() (nbodykit.core.datasource.DataSource method), 56
parallel_read() (nbodykit.core.datasource.FastPM.FastPMDataSource method), 62
parallel_read() (nbodykit.core.datasource.Gadget.GadgetDataSource method), 63
parallel_read() (nbodykit.core.datasource.HaloLabel.HaloLabel method), 65
parallel_read() (nbodykit.core.datasource.MultiFile.MultiFileDataSource method), 66
parallel_read() (nbodykit.core.datasource.TPMLabel.TPMLabel method), 75
parallel_read() (nbodykit.core.datasource.TPMSnapshot.TPMSnapshotData
method), 76

parallel_read() (nbodykit.core.datasource.ZeldovichSim.ZeldovichSimDataSource method), 78
parallel_read() (nbodykit.core.datasource.Zheng07HOD.Zheng07HodDataSource method), 80
parse_selection() (nbodykit.utils.selectionlanguage.Query method), 123
ParticleSource (class in nbodykit.core.source.Particle), 87
PlainTextDataSource (class in nbodykit.core.datasource.PlainText), 68
Play (class in nbodykit.core.algorithms.ExampleAlgorithm), 36
plugin_name (nbodykit.core.algorithms.BianchiFFTPower.BianchiPowerAlgorithm attribute), 35
plugin_name (nbodykit.core.algorithms.ExampleAlgorithm.Describe attribute), 36
plugin_name (nbodykit.core.algorithms.ExampleAlgorithm.Play attribute), 37
plugin_name (nbodykit.core.algorithms.FiberCollisions.FiberCollisionsAlgorithm attribute), 40
plugin_name (nbodykit.core.algorithms.FOF.FOFAlgorithm attribute), 38
plugin_name (nbodykit.core.algorithms.FOF6D.FOF6DAlgorithm attribute), 39
plugin_name (nbodykit.core.algorithms.PaintGrid.PaintGridAlgorithm attribute), 42
plugin_name (nbodykit.core.algorithms.PairCountCorrelation.PairCountCorrelationAlgorithm attribute), 43
plugin_name (nbodykit.core.algorithms.PeriodicBox.FFTCorrelationAlgorithm attribute), 46
plugin_name (nbodykit.core.algorithms.PeriodicBox.FFTPowerAlgorithm attribute), 48
plugin_name (nbodykit.core.algorithms.RedshiftHistogram.RedshiftHistogramAlgorithm attribute), 50
plugin_name (nbodykit.core.algorithms.Subsample.Subsample attribute), 51
plugin_name (nbodykit.core.algorithms.TestBoxSize.TestBoxSizeAlgorithm attribute), 52
plugin_name (nbodykit.core.algorithms.TidalTensor.TidalTensor attribute), 53
plugin_name (nbodykit.core.algorithms.TraceHalo.TraceHaloAlgorithm attribute), 54
plugin_name (nbodykit.core.datasource.BigFileGrid.BigFileGridSource attribute), 60
plugin_name (nbodykit.core.datasource.FastPM.FastPMDataSource attribute), 62
plugin_name (nbodykit.core.datasource.FOFGroups.FOFDataSource attribute), 61
plugin_name (nbodykit.core.datasource.Gadget.GadgetDataSource attribute), 63
plugin_name (nbodykit.core.datasource.Gadget.GadgetGroupTabDataSource attribute), 65
plugin_name (nbodykit.core.datasource.HaloLabel.HaloLabel attribute), 66
plugin_name (nbodykit.core.datasource.MultiFile.MultiFileDataSource attribute), 66
plugin_name (nbodykit.core.datasource.Pandas.PandasDataSource attribute), 68
plugin_name (nbodykit.core.datasource.PlainText.PlainTextDataSource attribute), 70
plugin_name (nbodykit.core.datasource.RaDecRedshift.RaDecRedshiftData attribute), 71
plugin_name (nbodykit.core.datasource.ShiftedObserver.ShiftedObserverDa attribute), 73
plugin_name (nbodykit.core.datasource.Subsample.SubsampleDataSource attribute), 74
plugin_name (nbodykit.core.datasource.TPMLabel.TPMLabel attribute), 75
plugin_name (nbodykit.core.datasource.TPMSnapshot.TPMSnapshotDataS attribute), 76
plugin_name (nbodykit.core.datasource.UniformBox.UniformBoxDataSour attribute), 77
plugin_name (nbodykit.core.datasource.ZeldovichSim.ZeldovichSimDataSo attribute), 78
plugin_name (nbodykit.core.datasource.Zheng07HOD.Zheng07HodDataSo attribute), 80
plugin_name (nbodykit.core.painter.DefaultPainter.DefaultPainter attribute), 83
plugin_name (nbodykit.core.painter.MomentumPainter.MomentumPainter attribute), 84
plugin_name (nbodykit.core.source.Grid.GridSource attribute), 87
plugin_name (nbodykit.core.source.Particle.ParticleSource attribute), 88
plugin_name (nbodykit.core.transfer.TransferFunction.AnisotropicCIC attribute), 89
plugin_name (nbodykit.core.transfer.TransferFunction.AnisotropicTSC attribute), 90
plugin_name (nbodykit.core.transfer.TransferFunction.CICWindow attribute), 90
plugin_name (nbodykit.core.transfer.TransferFunction.NormalizeDC attribute), 91
plugin_name (nbodykit.core.transfer.TransferFunction.RemoveDC attribute), 91
plugin_name (nbodykit.core.transfer.TransferFunction.TSCWindow attribute), 92
plugin_name (nbodykit.io.bigfile.BigFile attribute), 102
plugin_name (nbodykit.io.binary.BinaryFile attribute), 103
plugin_name (nbodykit.io.csv.CSVFile attribute), 105
plugin_name (nbodykit.io.stack.FileStack attribute), 106
plugin_name (nbodykit.io.tpm.TPMBinaryFile attribute), 108
PluginBase (class in nbodykit.plugins), 109
PluginBaseMeta (in module nbodykit.plugins), 110
PluginManager (class in nbodykit.plugins.manager), 112
PluginParsingError, 110
poisson_sample_to_points() (in module nbodykit.mockmaker), 150

populate_mock() (nbodykit.core.datasource.Zheng07HOD.Zheng07HodDataSource method), 80
Power1dDataSet (class in nbodykit.dataset), 132
Power2dDataSet (class in nbodykit.dataset), 133
prev() (nbodykit.distributedarray.LinearTopology method), 136
project_to_basis() (in module nbodykit.measurestats), 147

Q

Query (class in nbodykit.utils.selectionlanguage), 122

R

RaDecDataSource() (in module nbodykit.core.algorithms.FiberCollisions), 40
RaDecRedshiftDataSource (class in nbodykit.core.datasource.RaDecRedshift), 70
read() (nbodykit.core.datasource.BigFileGrid.BigFileGridSource method), 60
read() (nbodykit.core.datasource.DataStream method), 58
read() (nbodykit.core.datasource.Gadget.GadgetGroupTabDataSource method), 65
read() (nbodykit.core.datasource.GridSource method), 59
read() (nbodykit.core.GridSource method), 30, 97
read() (nbodykit.core.Source method), 27, 94
read() (nbodykit.core.source.Grid.GridSource method), 87
read() (nbodykit.core.source.Particle.ParticleSource method), 88
read() (nbodykit.core.source.Source method), 85
read() (nbodykit.files.GadgetGroupTabFile method), 137
read() (nbodykit.files.GadgetSnapshotFile method), 138
read() (nbodykit.files.Snapshot method), 139
read() (nbodykit.files.TPMSnapshotFile method), 140
read() (nbodykit.fkp.FKPCatalog method), 142
read() (nbodykit.io.bigfile.BigFile method), 102
read() (nbodykit.io.binary.BinaryFile method), 103
read() (nbodykit.io.csv.CSVFile method), 105
read() (nbodykit.io.FileType method), 101
read() (nbodykit.io.stack.FileStack method), 106
read() (nbodykit.stripedfile.DataStorage method), 155
read() (nbodykit.stripedfile.StripeFile method), 155
Read1DPlainText() (in module nbodykit.files), 138
Read2DPlainText() (in module nbodykit.files), 139
readall() (nbodykit.core.DataSource method), 29, 96
readall() (nbodykit.core.datasource.DataSource method), 57
readall() (nbodykit.core.datasource.FOFGroups.FOFDataSource method), 61
readall() (nbodykit.core.datasource.Pandas.PandasDataSource method), 68
readall() (nbodykit.core.datasource.PlainText.PlainTextDataSource method), 70
readall() (nbodykit.core.datasource.RaDecRedshift.RaDecRedshiftDataSou method), 71
readall() (nbodykit.core.datasource.Subsample.SubsampleDataSource method), 74
readall() (nbodykit.core.datasource.UniformBox.UniformBoxDataSource method), 77
readat() (nbodykit.stripedfile.StripeFile method), 156
ReadConfigFile() (in module nbodykit.plugins.fromfile), 110
RedshiftBins() (in module nbodykit.core.algorithms.RedshiftHistogram), 49
RedshiftHistogramAlgorithm (class in nbodykit.core.algorithms.RedshiftHistogram), 49
reindex() (nbodykit.dataset.DataSet method), 130
RemoveDC (class in nbodykit.core.transfer.TransferFunction), 91
rename_variable() (nbodykit.dataset.DataSet method), 131
replacesorted() (in module nbodykit.ndarray), 152
reprsymbol (nbodykit.utils.selectionlanguage.BoolAnd attribute), 120
reprsymbol (nbodykit.utils.selectionlanguage.BoolOr attribute), 121
required_attributes (nbodykit.core.Painter attribute), 31, 98
required_attributes (nbodykit.core.painter.Painter attribute), 81
required_attributes (nbodykit.io.FileType attribute), 101
RightCompOperand (class in nbodykit.utils.selectionlanguage), 123
run() (nbodykit.core.Algorithm method), 26, 93
run() (nbodykit.core.algorithms.Algorithm method), 32
run() (nbodykit.core.algorithms.BianchiFFTPower.BianchiPowerAlgorithm method), 35
run() (nbodykit.core.algorithms.ExampleAlgorithm.Describe method), 36
run() (nbodykit.core.algorithms.ExampleAlgorithm.Play method), 37
run() (nbodykit.core.algorithms.FiberCollisions.FiberCollisionsAlgorithm method), 40
run() (nbodykit.core.algorithms.FOF.FOFAlgorithm method), 38
run() (nbodykit.core.algorithms.FOF6D.FOF6DAlgorithm method), 39
run() (nbodykit.core.algorithms.PaintGrid.PaintGridAlgorithm method), 42
run() (nbodykit.core.algorithms.PairCountCorrelation.PairCountCorrelation method), 43
run() (nbodykit.core.algorithms.PeriodicBox.FFTCorrelationAlgorithm method), 46

run() (nbodykit.core.algorithms.PeriodicBox.FFTPowerAlgorithm method), 48
run() (nbodykit.core.algorithms.RedshiftHistogram.RedshiftHistogramAlgorithm method), 50
run() (nbodykit.core.algorithms.Subsample.Subsample method), 51
run() (nbodykit.core.algorithms.TestBoxSize.TestBoxSizeAlgorithm method), 52
run() (nbodykit.core.algorithms.TidalTensor.TidalTensor method), 53
run() (nbodykit.core.algorithms.TraceHalo.TraceHaloAlgorithm method), 54

S

sample() (nbodykit.cosmology.CosmologyBase method), 126
save() (nbodykit.core.Algorithm method), 26, 93
save() (nbodykit.core.algorithms.Algorithm method), 33
save() (nbodykit.core.algorithms.BianchiFFTPower.BianchiPowerAlgorithm method), 35
save() (nbodykit.core.algorithms.ExampleAlgorithm.Describe method), 36
save() (nbodykit.core.algorithms.ExampleAlgorithm.Play method), 37
save() (nbodykit.core.algorithms.FiberCollisions.FiberCollisionsAlgorithm method), 40
save() (nbodykit.core.algorithms.FOF.FOFAlgorithm method), 38
save() (nbodykit.core.algorithms.FOF6D.FOF6DAlgorithm method), 39
save() (nbodykit.core.algorithms.PaintGrid.PaintGridAlgorithm method), 42
save() (nbodykit.core.algorithms.PairCountCorrelation.PairCountCorrelationAlgorithm method), 43
save() (nbodykit.core.algorithms.PeriodicBox.FFTCorrelationAlgorithm method), 46
save() (nbodykit.core.algorithms.PeriodicBox.FFTPowerAlgorithm method), 48
save() (nbodykit.core.algorithms.RedshiftHistogram.RedshiftHistogramAlgorithm method), 50
save() (nbodykit.core.algorithms.Subsample.Subsample method), 51
save() (nbodykit.core.algorithms.TestBoxSize.TestBoxSizeAlgorithm method), 52
save() (nbodykit.core.algorithms.TidalTensor.TidalTensor method), 53
save() (nbodykit.core.algorithms.TraceHalo.TraceHaloAlgorithm method), 54
ScatterArray() (in module nbodykit.distributedarray), 137
schema (nbodykit.core.algorithms.BianchiFFTPower.BianchiPowerAlgorithm attribute), 35
schema (nbodykit.core.algorithms.ExampleAlgorithm.Describe attribute), 36
schema (nbodykit.core.algorithms.ExampleAlgorithm.Play attribute), 37
schema (nbodykit.core.algorithms.FiberCollisions.FiberCollisionsAlgorithm attribute), 40
schema (nbodykit.core.algorithms.FOF.FOFAlgorithm attribute), 38
schema (nbodykit.core.algorithms.FOF6D.FOF6DAlgorithm attribute), 39
schema (nbodykit.core.algorithms.PaintGrid.PaintGridAlgorithm attribute), 42
schema (nbodykit.core.algorithms.PairCountCorrelation.PairCountCorrelati attribute), 44
schema (nbodykit.core.algorithms.PeriodicBox.FFTCorrelationAlgorithm attribute), 46
schema (nbodykit.core.algorithms.PeriodicBox.FFTPowerAlgorithm attribute), 48
schema (nbodykit.core.algorithms.RedshiftHistogram.RedshiftHistogramAl attribute), 50
schema (nbodykit.core.algorithms.Subsample.Subsample attribute), 51
schema (nbodykit.core.algorithms.TestBoxSize.TestBoxSizeAlgorithm attribute), 52
schema (nbodykit.core.algorithms.TidalTensor.TidalTensor attribute), 53
schema (nbodykit.core.algorithms.TraceHalo.TraceHaloAlgorithm attribute), 54
schema (nbodykit.core.datasource.BigFileGrid.BigFileGridSource attribute), 60
schema (nbodykit.core.datasource.FastPM.FastPMDataSource attribute), 62
schema (nbodykit.core.datasource.FOFGroups.FOFDataSource attribute), 61
schema (nbodykit.core.datasource.Gadget.GadgetDataSource attribute), 63
schema (nbodykit.core.datasource.Gadget.GadgetGroupTabDataSource attribute), 65
schema (nbodykit.core.datasource.HaloLabel.HaloLabel attribute), 66
schema (nbodykit.core.datasource.MultiFile.MultiFileDataSource attribute), 67
schema (nbodykit.core.datasource.Pandas.PandasDataSource attribute), 68
schema (nbodykit.core.datasource.PlainText.PlainTextDataSource attribute), 70
schema (nbodykit.core.datasource.RaDecRedshift.RaDecRedshiftDataSourc attribute), 72
schema (nbodykit.core.datasource.ShiftedObserver.ShiftedObserverDataSou attribute), 73
schema (nbodykit.core.datasource.Subsample.SubsampleDataSource attribute), 74
schema (nbodykit.core.datasource.TPMLabel.TPMLabel attribute), 75
schema (nbodykit.core.datasource.TPMSnapshot.TPMSnapshotDataSource attribute), 76

schema (nbodykit.core.datasource.UniformBox.UniformBoxDataSource attribute), 77
schema (nbodykit.core.datasource.ZeldovichSim.ZeldovichSimDataSource attribute), 78
schema (nbodykit.core.datasource.Zheng07HOD.Zheng07HodDataSource attribute), 80
schema (nbodykit.core.painter.DefaultPainter.DefaultPainter attribute), 83
schema (nbodykit.core.painter.MomentumPainter.MomentumPainter attribute), 84
schema (nbodykit.core.source.Grid.GridSource attribute), 87
schema (nbodykit.core.source.Particle.ParticleSource attribute), 88
schema (nbodykit.core.transfer.TransferFunction.AnisotropicCIC attribute), 89
schema (nbodykit.core.transfer.TransferFunction.AnisotropicTSC attribute), 90
schema (nbodykit.core.transfer.TransferFunction.CICWindow attribute), 90
schema (nbodykit.core.transfer.TransferFunction.NormalizeDC attribute), 91
schema (nbodykit.core.transfer.TransferFunction.RemoveDC attribute), 91
schema (nbodykit.core.transfer.TransferFunction.TSCWindow attribute), 92
schema (nbodykit.io.bigfile.BigFile attribute), 102
schema (nbodykit.io.binary.BinaryFile attribute), 103
schema (nbodykit.io.csv.CSVFile attribute), 105
schema (nbodykit.io.stack.FileStack attribute), 106
schema (nbodykit.io.tpm.TPMBinaryFile attribute), 108
scotts_bin_width() (in module nbodykit.core.algorithms.RedshiftHistogram), 50
sel() (nbodykit.dataset.DataSet method), 131
SelectionError, 123
set() (nbodykit.GlobalComm class method), 92
set() (nbodykit.GlobalCosmology class method), 93
set_random_seed() (in module nbodykit.core.datasource.Zheng07HOD), 80
shape (nbodykit.dataset.DataSet attribute), 132
shape (nbodykit.io.FileType attribute), 101
shape (nbodykit.utils.meshtools.MeshSlab attribute), 119
ShiftedObserverDataSource (class in nbodykit.core.datasource.ShiftedObserver), 72
shiftedpaint() (nbodykit.core.Painter method), 31, 98
shiftedpaint() (nbodykit.core.painter.Painter method), 81
size (nbodykit.core.DataSource attribute), 29, 97
size (nbodykit.core.datasource.DataSource attribute), 57
SlabIterator() (in module nbodykit.utils.meshtools), 119
Smoothing() (nbodykit.core.algorithms.TidalTensor.TidalTensor method), 53
SnapshotSource (class in nbodykit.files), 139
so() (in module nbodykit.core.algorithms.FOF6D), 39
sort() (nbodykit.distributedarray.DistributedArray method), 135
Source (class in nbodykit.core), 26, 94
Source (class in nbodykit.core.source), 85
split_ranks() (in module nbodykit.utils.taskmanager), 125
split_size_3d() (in module nbodykit.fof), 144
squeeze() (nbodykit.dataset.DataSet method), 132
string (nbodykit.plugins.PluginBase attribute), 110
StripeFile (class in nbodykit.stripedfile), 155
subfof() (in module nbodykit.core.algorithms.FOF6D), 39
Subsample (class in nbodykit.core.algorithms.Subsample), 50
SubsampleDataSource (class in nbodykit.core.datasource.Subsample), 73
supported_types (nbodykit.plugins.manager.PluginManager attribute), 113

T

tails() (nbodykit.distributedarray.LinearTopology method), 137
TaskManager (class in nbodykit.utils.taskmanager), 124
test() (in module nbodykit.distributedarray), 137
TestBoxSizeAlgorithm (class in nbodykit.core.algorithms.TestBoxSize), 51
TidalTensor (class in nbodykit.core.algorithms.TidalTensor), 52
TidalTensor() (nbodykit.core.algorithms.TidalTensor.TidalTensor method), 53
timer() (in module nbodykit.utils), 117
TPMBinaryFile (class in nbodykit.io.tpm), 108
TPMLabel (class in nbodykit.core.datasource.TPMLabel), 74
TPMSnapshotDataSource (class in nbodykit.core.datasource.TPMSnapshot), 75
TPMSnapshotFile (class in nbodykit.files), 139
TraceHaloAlgorithm (class in nbodykit.core.algorithms.TraceHalo), 53
Transfer (class in nbodykit.core), 31, 98
Transfer (class in nbodykit.core.transfer), 88
transform() (nbodykit.core.datasource.ShiftedObserver.ShiftedObserverDat method), 73
transform() (nbodykit.core.source.Painter method), 84
TSCWindow (class in nbodykit.core.transfer.TransferFunction), 91

U

UniformBoxDataSource (class in nbodykit.core.datasource.UniformBox), 76

unique_labels() (nbodykit.distributedarray.DistributedArray method), 135
unsample() (nbodykit.cosmology.CosmologyBase method), 126
update_and_populate() (nbodykit.core.datasource.Zheng07HOD.Zheng07HodDataSource method), 80

V

validate__init__() (in module nbodykit.plugins.validate), 117
validate__init__signature() (in module nbodykit.plugins.validate), 117
validate_choices() (in module nbodykit.plugins.validate), 117
validate_required_attributes() (in module nbodykit.plugins.validate), 117
variables (nbodykit.dataset.DataSet attribute), 132

W

wait() (nbodykit.utils.taskmanager.TaskManager method), 125
write() (nbodykit.files.Snapshot method), 139
write() (nbodykit.files.TPMSnapshotFile method), 140
write() (nbodykit.storage.Measurement1DStorage method), 153
write() (nbodykit.storage.Measurement2DStorage method), 154
write() (nbodykit.storage.MeasurementStorage method), 154
write() (nbodykit.stripedfile.DataStorage method), 155
write() (nbodykit.stripedfile.StripeFile method), 156
write_hdf5() (nbodykit.core.algorithms.Subsample.Subsample method), 51
write_hdf5() (nbodykit.core.algorithms.TidalTensor.TidalTensor method), 53
write_mwhite_subsample() (nbodykit.core.algorithms.Subsample.Subsample method), 51
writeat() (nbodykit.stripedfile.StripeFile method), 156

Z

ZeldovichSimDataSource (class in nbodykit.core.datasource.ZeldovichSim), 77
Zheng07HodDataSource (class in nbodykit.core.datasource.Zheng07HOD), 78