Lawrence Berkeley National Laboratory
LBL Publications

Title
nbodykit: An Open-source, Massively Parallel Toolkit for Large-scale Structure

Permalink
https://escholarship.org/uc/item/8n00v7n0

Journal
The Astronomical Journal, 156(4)

ISSN
0004-6256

Authors
Hand, Nick
Feng, Yu
Beutler, Florian
et al.

Publication Date
2018-10-01

DOI
10.3847/1538-3881/aadae0

Peer reviewed

NBODYKIT: AN OPEN-SOURCE, MASSIVELY PARALLEL TOOLKIT FOR LARGE-SCALE STRUCTURE

Nick Hand,[1,2] Yu Feng,[2] Florian Beutler,[3,4] Yin Li,[2,5,4,6] Chirag Modi,[5,2] Uroš Seljak,[5,2] and
Zachary Slepian[2,4,*]

[1] *Astronomy Department, University of California, Berkeley, CA 94720, USA*
[2] *Berkeley Center for Cosmological Physics, University of California, Berkeley CA 94720*
[3] *Institute of Cosmology & Gravitation, Dennis Sciama Building, University of Portsmouth, Portsmouth, PO1 3FX, UK*
[4] *Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720, USA*
[5] *Physics Department, University of California, Berkeley, CA 94720, USA*
[6] *Kavli Institute for the Physics and Mathematics of the Universe (WPI), UTIAS, The University of Tokyo, Chiba 277–8583, Japan*

ABSTRACT

We present `nbodykit`, an open-source, massively parallel Python toolkit for analyzing large-scale structure (LSS) data. Using Python bindings of the Message Passing Interface (MPI), we provide parallel implementations of many commonly used algorithms in LSS. `nbodykit` is both an interactive and scalable piece of scientific software, performing well in a supercomputing environment while still taking advantage of the interactive tools provided by the Python ecosystem. Existing functionality includes estimators of the power spectrum, 2 and 3-point correlation functions, a Friends-of-Friends grouping algorithm, mock catalog creation via the halo occupation distribution technique, and approximate N-body simulations via the FastPM scheme. The package also provides a set of distributed data containers, insulated from the algorithms themselves, that enable `nbodykit` to provide a unified treatment of both simulation and observational data sets. `nbodykit` can be easily deployed in a high performance computing environment, overcoming some of the traditional difficulties of using Python on supercomputers. We provide performance benchmarks illustrating the scalability of the software. The modular, component-based approach of `nbodykit` allows researchers to easily build complex applications using its tools. The package is extensively documented at http://nbodykit.readthedocs.io, which also includes an interactive set of example recipes for new users to explore. As open-source software, we hope `nbodykit` provides a common framework for the community to use and develop in confronting the analysis challenges of future LSS surveys.

Corresponding author: Nick Hand
nhand@berkeley.edu

* Einstein Fellow

1. INTRODUCTION

The analysis of LSS data sets has played a pivotal role in establishing the current concordance paradigm in modern cosmology, the ΛCDM model. From the earliest galaxy surveys (Davis et al. 1985; Maddox et al. 1990), comparisons between the theoretical predictions for the distribution of matter in the Universe and observations have proven to be a valuable tool. Indeed, LSS observations, in combination with cosmic microwave background (CMB) measurements, provided some of the earliest evidence for the ΛCDM model, e.g., Efstathiou et al. (1990); Krauss & Turner (1995); Ostriker & Steinhardt (1995). Interest in LSS surveys increased immensely following the first direct evidence for cosmic acceleration (Riess et al. 1998; Perlmutter et al. 1999), as it was realized that the baryon acoustic oscillation (BAO) feature imprinted on large-scale clustering provided a "standard ruler" to map the expansion history (Eisenstein et al. 1998; Blake & Glazebrook 2003; Seo & Eisenstein 2003). From its first measurements (Cole et al. 2005; Eisenstein et al. 2005) to more recent studies (Font-Ribera et al. 2014; Delubac et al. 2015; Alam et al. 2017; Slepian et al. 2017), the BAO has proved to be a valuable probe of cosmic acceleration, enabling the most precise measurements of the expansion history of the Universe over a wide redshift range. Analyses of these data sets have also pushed us closer to answering other important questions in contemporary cosmology, including deviations from General Relativity (Mueller et al. 2016), the neutrino mass scale (Lesgourgues & Pastor 2006; Beutler et al. 2014), and the existence of primordial non-Gaussianity (Slosar et al. 2008; Desjacques & Seljak 2010).

The foundations of the numerical methods used in LSS data analysis today go back several decades. Hockney & Eastwood (1981) discussed several important computer simulation methods, including but not limited to mass assignment interpolation windows and the interlacing technique for reducing aliasing. The Friends-of-Friends (FOF) algorithm for identifying halos from a numerical simulation was first utilized in Davis et al. (1985). The most commonly used clustering estimators for the two-point correlation function (2PCF) and power spectrum were first developed in Landy & Szalay (1993) and Feldman et al. (1994), respectively, and techniques to measure anisotropic clustering via a multipole basis were first used around the same time, e.g., Cole et al. (1995). Other modern, well-established numerical techniques include N-body simulation methods, e.g., Springel et al. (2001); Springel (2005), and the use of KD-trees in correlation function estimators (Moore et al. 2001).

Recent years have brought important updates to these analysis techniques. Advances in LSS observations, with increased sample sizes and statistical precision, have driven the development of new statistical estimators, while also increasing modeling complexities and creating a need to reduce wall-clock times. Recently, we have seen faster power spectrum and 2PCF multipole estimators (Yamamoto et al. 2006; Scoccimarro 2015; Bianchi et al. 2015; Slepian & Eisenstein 2015a, 2016; Hand et al. 2017a) and improved FOF algorithms (Springel 2005; Behroozi et al. 2013; Feng & Modi 2017). Highly optimized software, e.g., `Corrfunc` (Sinha & Garrison 2017), is also becoming increasingly common. New statistical estimators, e.g., Slepian & Eisenstein (2015b, 2017); Castorina & White (2017), are being developed to extract as much information as possible from LSS surveys. The rise of particle mesh simulation methods (Merz et al. 2005; Tassev et al. 2013; White et al. 2014; Feng et al. 2016) has offered a computationally cheaper alternative to running full N-body simulations. Finally, tools have emerged to help deal with the growing complexities of modeling the connection between halos and galaxies (Hearin et al. 2017). These examples represent just a sampling of the recent updates to LSS data analysis and modeling techniques.

The well-established foundation of LSS numerical methods suggests the community could benefit from a standard software package providing implementations of these methods. Such a package would also serve as a common framework for users as they incorporate future extensions and advancements. Given the already rising wall-clock times of current analyses and the expected volume of data from next-generation LSS surveys, scaling performance should also be a key priority.

Several computing trends in the past few years have emerged to help make such a software package possible. First, the Python programming language[1] has emerged as the most popular language in the field of astronomy (Momcheva & Tollerud 2015; NSF 2017), and the `astropy`[2] package (Astropy Collaboration et al. 2013) has led the development of an astronomy-focused Python ecosystem. Python's elegant syntax and dynamic nature make the language easy to learn and work with. Combined with its object-oriented focus and the larger ecosystem containing `SciPy`[3] (Jones et al. 2001–2017), `NumPy`[4] (van der Walt et al. 2011), `IPython`[5]

---

[1] http://python.org
[2] http://www.astropy.org
[3] https://www.scipy.org
[4] http://www.numpy.org
[5] https://ipython.org

(Perez & Granger 2007), and `Jupyter`[6] (Thomas et al. 2016), Python is well-suited for both rapid application development and use in scientific research. Second, the availability and performance of large-scale computing resources continues to grow, and initiatives, e.g., The Exascale Computing Project,[7] have been established to ensure the sustainability of this trend. At the same time, solutions to the traditional barriers to using Python on massively parallel, high-performance computing (HPC) machines have been developed. The `mpi4py` package (Dalcín et al. 2008; Dalcin et al. 2011) has facilitated the development of parallel Python applications by providing bindings of the Message Passing Interface (MPI) standard. Furthermore, tools have been developed, e.g., Feng & Hand (2016), to alleviate the start-up bottleneck encountered when launching Python applications on HPC systems.

Motivated by these recent developments, we present the first public release of `nbodykit` (v0.3.0), an open-source, parallel toolkit written in Python for use in the analysis of LSS data. Designed for use on HPC machines, `nbodykit` includes fully parallel implementations of a canonical set of LSS algorithms. It also includes a set of distributed and extensible data containers, which can support a wide variety of data formats and large volumes of data. These data containers are insulated from the algorithms themselves, allowing `nbodykit` to be used for either simulation or observational data sets. We have balanced the scalable nature of `nbodykit` with an object-oriented, component-based design that also facilitates interactive use. This allows researchers to take advantage of interactive Python tools, e.g., the `Jupyter` notebook, as well as integrate `nbodykit` components with their own software to build larger applications that solve specific problems in LSS.

`nbodykit` has been developed, tested, and deployed on the Edison and Cori Cray supercomputers at the National Energy Research Scientific Computing Center (NERSC) and has been utilized in several published research studies (Hand et al. 2017a,b; Ding et al. 2017; Pinol et al. 2017; Schmittfull et al. 2017; Modi et al. 2016; Feng et al. 2016; Waters et al. 2016). Since its start, it has been developed on GitHub as open-source software at https://github.com/bccp/nbodykit.

The objective of this paper is to provide an overview of the `nbodykit` software and familiarize the community with some of its capabilities. We hope that researchers find `nbodykit` to be a useful tool in their sci-

entific work and in the spirit of open science, that it continues to grow via community contributions. Extensive documentation and tutorials are available at http://nbodykit.readthedocs.io, and we do not aim to provide such detailed documentation in this work. The documentation also includes instructions for launching an interactive environment containing a set of example recipes. This allows new users to explore `nbodykit` without setting up their own `nbodykit` installation.

The paper is organized as follows. We provide a broad overview of `nbodykit` in Section 2 and discuss a more detailed list of its capabilities in Section 3. We describe our development process and deployment strategy for `nbodykit` in Section 4. Section 5 presents an illustrative example use case, and Section 6 outlines performance benchmarks for various algorithms. Finally, we conclude and summarize in Section 7.

2. OVERVIEW

2.1. *Initializing* `nbodykit`

A core design goal of `nbodykit` is maintaining an interactive user experience, allowing the user to quickly experiment and to prototype new analysis pipelines while still leveraging the power of parallel processing when necessary. We adopt a "lab" framework for `nbodykit`, where all of the necessary data containers and algorithms can be imported from the `nbodykit.lab` module. Furthermore, we utilize Python's `logging` module to print messages at runtime, which allows users to track the progress of the application in real time. Typically, applications using `nbodykit` begin with the following statements:

```
from nbodykit.lab import *
from nbodykit import setup_logging

setup_logging()
```

2.2. *The* `nbodykit` *Ecosystem*

`nbodykit` is explicitly maintained as a pure Python package. However, it depends on several compiled extension packages that each focus on more specialized tasks. This approach enables `nbodykit` to describe higher-level abstractions in Python and retains the readability, syntax, and user interface benefits of the Python language. For computationally expensive sections of the code base, we use the compiled extension packages for speed. With the emergence of Python package managers such as Anaconda,[8] the availability of binary versions of these com-

---

[6] http://jupyter.org
[7] https://www.exascaleproject.org

[8] https://anaconda.com

piled packages for different operating systems has suffi-
ciently eased most installation issues in our experience
(see Section 4.3).

Below, we describe some of the more important depen-
dencies of `nbodykit`, each of which is focused on solving
a particular problem:

- `pfft-python`: a Python binding of the `PFFT` soft-
  ware (Pippig 2013), which computes parallel fast
  Fourier transforms (FFTs) (Feng 2017a).

- `pmesh`: particle mesh calculations, including den-
  sity field interpolation and discrete parallel FFTs
  via `pfft-python` (Feng 2017b).

- `bigfile`: a reproducible, massively parallel in-
  put/output (IO) library for large, hierarchical data
  sets (Feng 2017c).

- `kdcount`: spatial indexing operations via KD-trees
  (Feng 2017d).

- `classylss`: a Python binding of the `CLASS` Boltz-
  mann solver (Hand & Feng 2017).

- `fastpm-python`: a Python implementation of the
  FastPM scheme for quasi N-body simulations
  (Feng 2017e; Feng et al. 2016).

- `Corrfunc`: a set of high-performance routines for
  computing pair counting statistics (Sinha & Gar-
  rison 2017).

- `Halotools`: a package to build and test models of
  the galaxy-halo connection (Hearin et al. 2017).

2.3. A Component-Based Approach

The design of `nbodykit` focuses on a modular,
component-based approach. The *components* are ex-
posed to the user as a set of Python classes and func-
tions, and users can combine these components to build
their specific applications. This design differs from the
more commonly used alternative in cosmology software,
which is a monolithic application controlled by a single
configuration file, e.g., as in `CAMB` (Lewis et al. 2000),
`CLASS` (Blas et al. 2011), and `Gadget` (Springel et al.
2001). We note that modular, object-oriented designs
using Python are becoming more popular recently, e.g.,
`astropy`, the `yt project` (Turk et al. 2011), `Halotools`
(Hearin et al. 2017), and Colossus (Diemer 2017). Dur-
ing the development process, we have found that a
component-based approach offers greater freedom and
flexibility to build complex applications with `nbodykit`.

We present some of the main classes and interfaces
and how data flows through them in Figure 1. In the
subsections to follow, we provide an overview of some of
the components outlined in this figure.

2.3.1. Catalog

A Catalog is a Python object derived from a
`CatalogSource` class that holds information about dis-
crete objects[9] in a columnar format. Catalogs imple-
ment a *random-read interface*, which allows users to
access arbitrary slices of the data while also taking ad-
vantage of the high throughput of a parallel file system.
Often, users will initialize Catalog objects by reading
data from a file on disk, using a `NumPy` array already
stored in memory, or by generating simulated particles
at runtime using one of `nbodykit`'s built-in classes.

2.3.2. Mesh

A Mesh is a Python object that computes a discrete
representation of a continuous quantity on a uniform
mesh. It is derived from a `MeshSource` class and pro-
vides a *paintable* interface, which refers to the process
of "painting" the density field values onto the discrete
mesh cells. When the user calls the `paint()` function,
the mesh data is returned as a three-dimensional array.
Mesh objects can be created directly from a Catalog via
the `to_mesh()` function or by generating simulated fields
directly on the mesh.

2.3.3. Algorithms

Algorithms are implemented as Python classes and
interact with data by consuming Catalog and Mesh ob-
jects as input. The algorithm is executed when the user
initializes the class, and the returned instance stores the
results as attributes.

2.3.4. Serialization and Reproducibility

Most objects in `nbodykit` are serializable[10] via a
`save()` function. Algorithm classes not only save the re-
sult of the algorithm but also save input parameters and
meta-data. They typically implement both a `save()`
and `load()` function, such that the algorithm result can
be de-serialized into an object of the same type. The two
main data containers, catalogs and meshes, can be seri-
alized using `nbodykit`'s intrinsic format which relies on
the massively parallel IO library `bigfile` (Feng 2017c).
`nbodykit` includes support for reading these serialized
results from disk back into Catalog or Mesh objects.

2.4. Parallelism
2.4.1. Data-based

---

[9] Here, "object" can represent galaxies, simulation particles,
mass elements, etc.

[10] Serialization (and its reverse, de-serialization) refers to the
process of storing a Python object on disk in a format such that
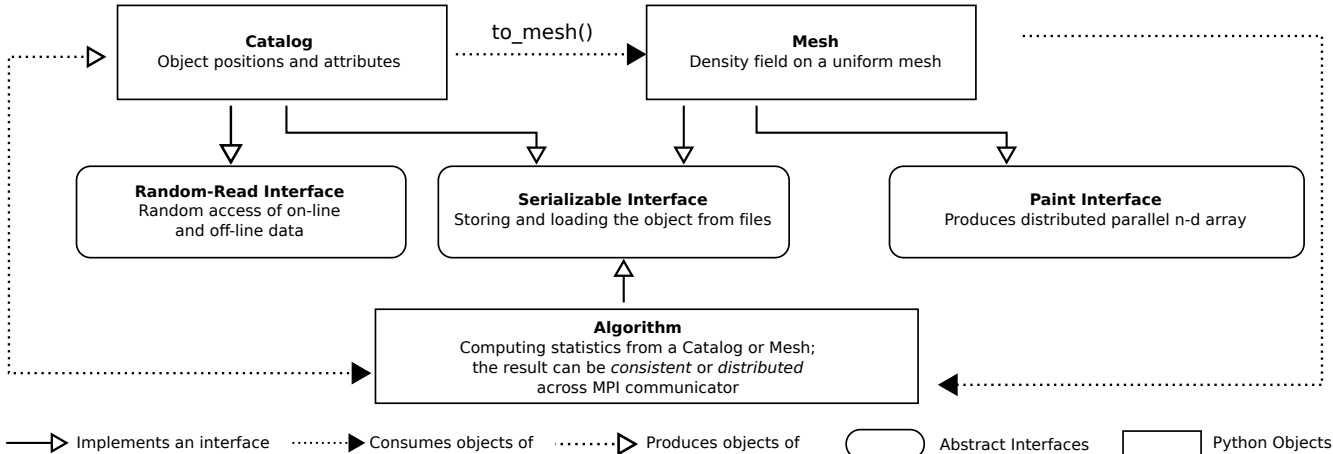it can be reconstructed at a later time.

Figure 1. The components and interfaces of `nbodykit`. The main Python classes are Catalog, Mesh, and Algorithm objects, which are described in more detail in §2.3. Algorithm results can be *consistent*, where all processes hold the same data, or *distributed*, where data is spread out evenly across parallel processes.

`nbodykit` is fully parallelized using the Python bindings of the MPI standard available through `mpi4py`. The MPI standard allows processes running in parallel, each with their own memory, to exchange messages. This mechanism enables independent results to be computed by individual processes and then combined into a single result.

Both the Catalog and Mesh objects are *distributed* data containers, meaning that the data is spread out evenly across the available processes within an MPI communicator.[11] Nearly all algorithm calculations are performed on this distributed data, with final results computed via a reduce operation across all processes in the communicator. Rarely throughout the code base, data is instead gathered to a single root process, and operations are performed on this data before re-distributing the results to all processes. This only occurs when wall-clock time will not be a concern for most use cases and the additional complexity of a massively parallel implementation is not merited.

The distributed nature of the Catalog object is implemented by using the random-read interface to access different slices of the tabular data for different processes. The values of a Mesh object are stored internally on a three-dimensional `NumPy` array, which is distributed evenly across all processes. The domain of the 3D mesh is decomposed across parallel processes using the particle mesh library `pmesh`, which also provides an interface for computing parallel FFTs of the mesh data using `pfft-python`. The `pfft-python` software exhibits ex-

cellent scaling with the number of available processes, enabling high-resolution (large number of cells) mesh calculations.

2.4.2. *Task-based*

The analysis of LSS data often involves hundreds to thousands of repetitions of a single, less computationally expensive task. Examples include estimating the covariance matrix of a clustering statistic from a set of simulations and best-fit parameter estimation for a model. `nbodykit` implements a `TaskManager` utility to allow users to easily iterate over multiple tasks while executing in parallel. Users can specify the desired number of processes assigned to each task, and the `TaskManager` will iterate through the tasks, ensuring that all processes are being utilized.

3. CAPABILITIES

In this section, we provide a more detailed overview of some of the main components of `nbodykit`. In particular, we describe how cosmology calculations are performed (§3.1), outline the available Catalog (§3.2) and Mesh (§3.3) classes, and provide details and references for the various algorithms currently implemented (§3.4).

3.1. *Cosmology*

The `nbodykit.cosmology` module includes functionality for representing cosmological parameter sets and computing various common theoretical quantities in LSS that depend on the background cosmological model. The underlying engine for these calculations is the `CLASS` Boltzmann solver (Blas et al. 2011; Lesgourgues 2011). We use the Python bindings of the `CLASS` C library provided by the `classylss` package. Comparing to the

---

[11] The MPI communicator is responsible for managing the communication between a set of parallel processes.

binding provided by the `CLASS` source code, `classylss` is a direct mapping of the `CLASS` object model to Python and integrates with the `NumPy` array protocol natively.

The main object in the module is the `Cosmology` class, which users can initialize by specifying a unique set of cosmological parameters (using the syntax of `CLASS`). This class represents the background cosmological model and contains methods to compute quantities that depend on the model. Most of the `CLASS` functionality is available through methods of the `Cosmology` object. Examples include distance as a function of redshift z, the Hubble parameter $H(z)$, the linear power spectrum, the nonlinear power spectrum, and the density and velocity transfer functions. Several `Cosmology` objects are provided for well-known parameter sets, including the *WMAP* 5, 7, and 9-year results (Komatsu et al. 2009, 2011; Hinshaw et al. 2013) and the *Planck* 2013 and 2015 results (Planck Collaboration et al. 2014, 2016).

The `nbodykit.cosmology` module also includes classes to represent theoretical power spectra and correlation functions. The `LinearPower` class can compute the linear power spectrum as a function of redshift and wavenumber, using either the transfer function as computed by `CLASS` or the analytic approximations of Eisenstein & Hu (1999). The latter includes the so-called "no-wiggle" transfer function, which includes no BAO but the correct broadband features and is useful for quantifying the significance of potential BAO features. Similarly, we provide the `NonlinearPower` object to compute nonlinear power spectra, using the `Halofit` implementation in `CLASS` (Smith et al. 2003), which includes corrections from Takahashi et al. (2012). The `ZeldovichPower` class uses the linear power spectrum object to compute the power spectrum in the Zel'dovich approximation (tree-level Lagrangian perturbation theory). The implementation closely follows the appendices of Vlah et al. (2015) and relies on a Python implementation and generalization of the `FFTLog` algorithm[12] (Hamilton 2000). Finally, we also provide a `CorrelationFunction` object to compute theoretical correlation functions from any of the available power classes (using `FFTLog` to compute the Fourier transform).

We choose to use the `CLASS` software for the cosmological engine in `nbodykit` rather than the most likely alternative, the `astropy.cosmology` module. This allows `nbodykit` to leverage the full power of a Boltzmann solver for LSS calculations. We provide syntax compatibility between the `Cosmology` class and `astropy`

when appropriate and provide functions to transform between the cosmology classes used by the two packages. However, we note that there are important differences between the two implementations. In particular, the treatment of massive neutrinos differs, with `astropy` using the approximations of Komatsu et al. (2011) rather than the direct calculations, as in `CLASS`.

3.2. *Catalogs*

In this section, we describe the two main ways that catalogs are created in `nbodykit`, as well as tools for cleaning and manipulating data stored in Catalog objects.

3.2.1. *Reading Data from Disk*

We provide support for loading data from disk into Catalog objects for several of the most common data storage formats in LSS data analysis. These formats include plaintext comma-separated value (CSV) data (via `pandas`, McKinney 2010), binary data stored in a columnar format, HDF5 data (via `h5py`, Collette & contributors 2017), FITS data (via `fitsio`, Sheldon 2017), and the `bigfile` data format. We also provide more specialized readers for particle data from the Tree-PM simulations of White (2002) and the legacy binary format of the GADGET simulations (Springel 2005). These Catalog objects use the `nbodykit.io` module, which includes several "file-like" classes for reading data from disk. These file-like objects implement a `read()` function that provides the random-read interface which returns a slice of the data for the requested columns. Users can easily support custom file formats by implementing their own subclass and `read()` interface.

Formats storing data on disk in a columnar format yield the best performance results, as the entirety of the data does not need to be parsed to yield the desired slice of the data on a given process. This is not true for the CSV storage format. We mitigate performance issues by implementing an enhanced version of the CSV parser in `pandas` that supports faster parallel random access. Our preferred IO format, `bigfile`, is massively parallel and stores data via a columnar format.

Finally, the Catalog object supports loading data from multiple files at once, providing a continuous view of the entirety of the data. This becomes particularly powerful when combined with the random-read interface, as arbitrary slices of the combined data can be accessed. For example, a single Catalog object can provide access to arbitrary slices of the output binary snapshots from an N-body simulation (stored over multiple files), often totaling 10-100 GB in size.

3.2.2. *Generating Catalogs at Runtime*

---

[12] https://github.com/eelregit/mcfit

nbodykit includes several Catalog classes that generate simulated data at runtime. The simplest of these allows generating random columns of data in parallel using the `numpy.random` module. We also provide a `UniformCatalog` class that generates uniformly distributed particles in a box. These classes are useful for testing purposes, as well as for use as unclustered, synthetic data in clustering estimators.

nbodykit also includes functionality for generating more realistic approximations of large-scale structure. `LogNormalCatalog` generates a set of objects by Poisson sampling a log-normal density field and applies the Zeldovich approximation to model nonlinear evolution (Coles & Jones 1991; Agrawal et al. 2017). The user can specify the input linear power spectrum and the desired output redshift of the catalog.

Catalog objects can also be created using the mock generation techniques of the `Halotools` software (Hearin et al. 2017) for populating halos with objects. `Halotools` includes functionality for populating halos via a wide range of techniques, including the halo occupation distribution (HOD), conditional luminosity function, and abundance matching methods. We refer the reader to Hearin et al. (2017) for further details. nbodykit supports using a generic `Halotools` model to populate a halo catalog. We also include built-in, specialized support for the HOD models of Zheng et al. (2007), Leauthaud et al. (2011), and Hearin et al. (2016).

Finally, the `fastpm-python` package implements an nbodykit Catalog object that generates particles via the FastPM approximate N-body simulation scheme (Feng et al. 2016). The FastPM library is massively parallel and exhibits excellent strong scaling with the number of available processes (see §6).

3.2.3. *On-demand Data Cleaning*

nbodykit uses the `dask` library (Dask Development Team 2016) to represent the data columns of a Catalog object as `dask` array objects instead of using the more familiar `NumPy` array. The `dask` array has two key features that help users work interactively with data, and, in particular, large data sets. The first feature is *delayed evaluation*. When manipulating a `dask` array, operations are not evaluated immediately but instead stored in a task graph. Users can explicitly evaluate the `dask` array (returning a `NumPy` array) via a call to a `compute()` function. Second, `dask` arrays are *chunked*. The array object is internally divided into many smaller arrays, and calculations are performed on these smaller "chunks."

The delayed evaluation of `dask` arrays is particularly useful during the process of data cleaning, when users manipulate input data before feeding it into the analysis pipeline. Common examples of data cleaning include changing the coordinate system from galactic to Euclidean, converting between unit conventions, and applying masks. When using large data sets, the time to load the full data set into memory can be significant. This delay hinders data exploration and limits the interactive benefits of the Python language. `dask` arrays allow users to design data-cleaning pipelines on the fly. If the data format on disk supports random-read access, users can easily select and peek at a small subset of data without reading the full data set. This becomes especially useful when prototyping scientific models in an interactive environment, such as a `Jupyter` notebook.

The chunked nature of the `dask` array allows array computations to be performed on large data sets that do not fit into memory because the chunk size defines the amount of data loaded into memory at any given time. It effectively extends the maximum size of useable data sets from the size of memory to the size of the disk storage. This feature also simplifies the process of dealing with large data sets in interactive environments.

3.3. *Meshes*
3.3.1. *Painting a Mesh*

The Mesh object implements a `paint()` function, which is responsible for generating the field values on the mesh and returning an array-like object to the user. Meshes provide an equal treatment of configuration and Fourier space, and users can specify whether the painted array is defined in configuration or Fourier space. In the former case, a `RealField` is returned and in the latter, a `ComplexField`. These objects are implemented by the `pmesh` package and are subclasses of the `NumPy ndarray` class. They are related via a real-to-complex parallel FFT operation, implemented using `pfft-python` via the `r2c()` and `c2r()` functions.

The `paint()` function paints mass-weighted (or equivalently, number-weighted) quantities to the mesh. The field that is painted is

$$F(\boldsymbol{x}) = [1 + \delta'(\boldsymbol{x})] V(\boldsymbol{x}), \qquad (1)$$

where $V(\boldsymbol{x})$ represents the field value painted to the mesh and $\delta'(\boldsymbol{x}) = n'(\boldsymbol{x})/\bar{n}' - 1$ is the weighted overdensity field. It is related to the unweighted number density as $n'(\boldsymbol{x}) = W(\boldsymbol{x})n(\boldsymbol{x})$, where $W(\boldsymbol{x})$ are the weights.

In nbodykit, users can control the behavior of both $V(\boldsymbol{x})$ and $W(\boldsymbol{x})$. In the default case, both quantities are unity, and the field painted to the mesh is $1 + \delta$. As an illustration, $V(\boldsymbol{x})$ can be specified as a velocity component to paint the momentum field (mass-weighted velocity). We also provide a mechanism by which users

can further transform the painted field on the mesh. The `apply()` function can be used to apply a function to the mesh, either in configuration or Fourier space. Multiple functions can be applied to the mesh, and the operations are performed when `paint()` is called.

3.3.2. *From Catalog to Mesh*

All Catalog objects include a `to_mesh()` function which creates a Mesh object using the specified number of cells per mesh side. This function allows users to configure exactly how the catalog is interpolated onto the mesh. Users can choose from several different mass assignment windows, including the Cloud-In-Cell (CIC), Triangular Shaped Cloud (TSC), and Piecewise Cubic Spline (PCS) schemes (Hockney & Eastwood 1981). The Daubechies wavelet (Daubechies 1992) and its symmetric counterpart ("Symlets", see, e.g., `PyWavelets`[13]) are also available. By default, the CIC window is used. The interlacing technique (Hockney & Eastwood 1981; Sefusatti et al. 2016) can reduce the effects of aliasing in Fourier space. In this scheme, the Catalog object is interpolated onto two separate meshes separated by half of a cell size. When the fields are combined in Fourier space, the leading-order contribution to aliasing is eliminated.

Users can also configure whether or not the window is *compensated*, which divides the density field in Fourier space by (Hockney & Eastwood 1981)

$$W(\boldsymbol{k}) = \Pi_i \left[\operatorname{sinc}\left(\pi k_i/2k_{\mathrm{N}}\right)\right]^p , \qquad (2)$$

where $i \in \{x, y, z\}$, $p = 2, 3, 4$ for CIC, TSC, and PCS, respectively, and $\operatorname{sinc}(x) \equiv \sin(x)/x$. The Nyquist frequency of the mesh is given by $k_{\mathrm{N}} = \pi N/L$, where L is the box size, and N is the number of cells per box side.

We provide comparisons of the various interpolation windows and correction methods in this section. First, Figure 2 illustrates the effects of interlacing when using the CIC, TSC, and PCS schemes. This comparison is similar to the detailed analysis presented in Sefusatti et al. (2016). Second, we show the effectiveness of the wavelet windows at reducing aliasing in Figure 3. For both figures, we paint a `LogNormalCatalog` of 5×10^7 objects to a mesh of 512^3 cells in a box of side length $2500 \ h^{-1}\mathrm{Mpc}$. We compare the measured power spectrum to a "reference" power spectrum, computed using a mesh of 1024^3 cells and the PCS window. When using the CIC, TSC, and PCS windows, we de-convolve the interpolation window using equation 2, while we apply no such corrections when using wavelet-based windows.
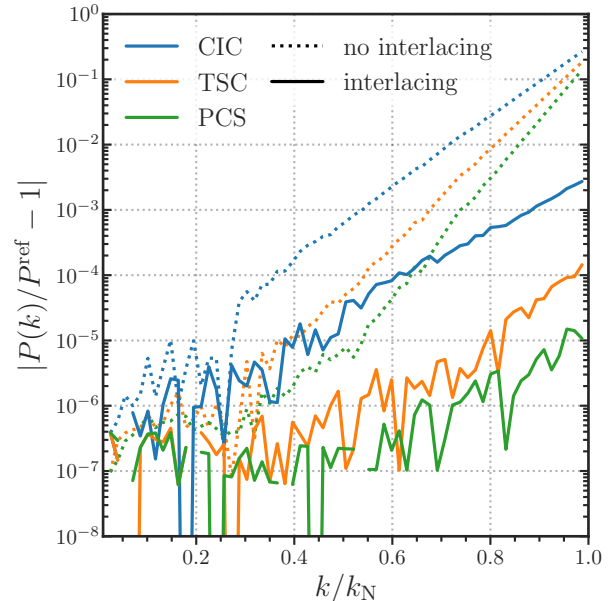
Figure 2. A comparison of the effects of interlacing when using the CIC, TSC, and PCS windows. We show the ratio of the power spectrum computed for a log-normal density field using a mesh with 512^3 cells to a reference power spectrum P^{ref}, computed using a mesh with 1024^3 cells. The ratio is shown as a function of wavenumber in units of the Nyquist frequency of the lower-resolution mesh. In all cases, the appropriate window compensation is performed using equation 2.

Figure 2 confirms the results of Sefusatti et al. (2016)—the interlacing technique performs very well at reducing the effects of aliasing on the measured power spectrum. We achieve sub-percent accuracy up to the Nyquist frequency when combining interlacing with the CIC, TSC, and PCS windows. In general, higher-order windows perform better, with the PCS scheme achieving a precision of at least $\sim 10^{-5}$ up to the Nyquist frequency.

Figure 3 compares the performance of the Daubechies and Symlet wavelets to the CIC, TSC, and PCS windows. As in Figure 2, we plot the ratio of the power spectrum computed using meshes of size 512^3 and 1024^3 cells. We apply equation 2 for the CIC, TSC, and PCS windows but do not apply any corrections when using the wavelet windows. For this comparison, we do not use interlacing. We are able to confirm the results of Cui et al. (2008) and Yang et al. (2009), which claim 2% accuracy on the power spectrum up to $k \approx 0.7k_{\mathrm{N}}$ when using the DB6 window without any additional corrections. However, the wavelet windows fail to match the precision achieved when using interlacing, even when using the largest wavelet size tested here ($a = 20$). Furthermore, the Daubechies windows introduce scale-

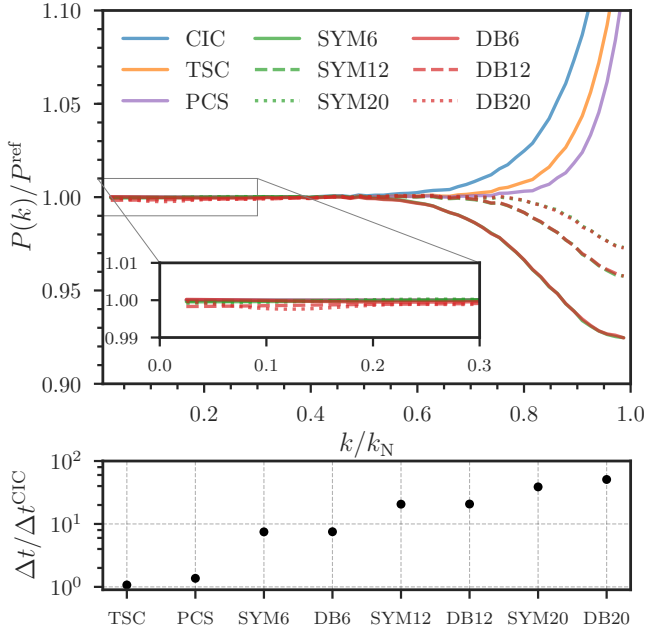**Figure 3.** The performance of the Daubechies and Symlet wavelets in comparison to the CIC, TSC, and PCS windows. Wavelet windows of sizes $a = 6$, 12, and 20 are shown. *Top*: the ratio of the measured power to the reference power spectrum, as in Figure 2. Here, we apply no corrections when using the wavelet windows and apply equation 2 for the CIC, TSC, and PCS windows. No interlacing is used for this test. *Bottom*: the speed of each interpolation window, relative to the CIC window. Speeds were recorded when computing the power spectra in the top panel.

dependence on large scales due to symmetry breaking (see the inset of Figure 3). The symmetric Symlet wavelets do not suffer from this issue but also cannot match the accuracy achieved when using interlacing.

Figure 3 also displays the relative speeds of each of the windows discussed in this section (bottom panel). These timing tests were performed using 64 processes on the NERSC Cori Phase I system. The wavelet windows are all significantly slower than the CIC, TSC, and PCS windows. The TSC and PCS methods are only marginally slower than the default CIC scheme, with slowdowns of ~10% and ~40%, respectively. The CIC, TSC, and PCS windows rely on optimized implementations in `pmesh`, while the wavelet windows use a slower lookup table implementation. Due to the precision of the interlacing technique and the relative speed of the TSC and PCS windows, we recommend using these options in most instances. However, it is generally best to determine the optimal set of parameters for a particular application by running convergence tests with different parameter configurations.

3.3.3. *An Illustrative Example*

```python
from nbodykit.lab import *
import matplotlib.pyplot as plt

# Initialize linear power spectrum with Planck 2015 cosmology
cosmo = cosmology.Planck15
Plin = cosmology.LinearPower(cosmo, redshift=0)

# Create a Catalog by sampling a log-normal density field
cat = LogNormalCatalog(Plin, nbar=3e-3, BoxSize=1380, Nmesh=256)

# Convert to a Mesh and use TSC painting
mesh = cat.to_mesh(Nmesh=256, window="tsc")

# Save the configuration-space Mesh
mesh.save("lognormal-mesh.bigfile", mode="real", dataset="Field")

# Preview a low-resolution projection of the density field
density = mesh.preview(Nmesh=64, axes=(0,1))
plt.imshow(density)

...

# Reload the Mesh from disk
mesh = BigFileMesh("lognormal-mesh.bigfile", dataset="Field")
```
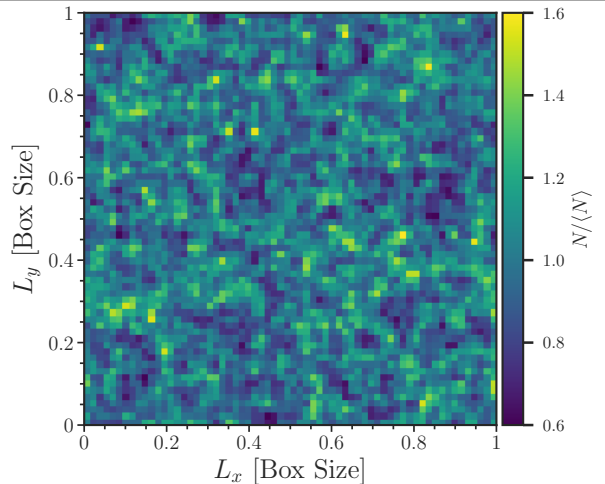

Figure 4. *Top*: an analysis pipeline illustrating the creation of a Mesh object from a Catalog, as well as how to serialize the painted mesh to disk and preview a low-resolution projection of the density field for inspection. *Bottom*: the two-dimensional, low-resolution preview of the painted density field $N/\langle N \rangle = 1 + \delta$.

We demonstrate the use of Mesh objects by example in Figure 4, which gives a short code snippet that creates a Mesh object from an existing Catalog, saves the configuration space density field to disk, and then reloads the data into memory. The snippet also demonstrates the `preview()` function, which can create a lower resolution projection of the full mesh field. This can be useful to quickly inspect mesh fields interactively, which would otherwise be difficult due to memory limitations. We show the preview of the density field from a lognormal catalog in the bottom panel of Figure 4, where the large-scale structure is clearly evident, even in the low-resolution projection.

3.4. *Algorithms*

The `nbodykit.algorithms` module includes parallel implementations of some of the most commonly used large-scale structure analysis algorithms. We take care to provide support for data sets from both observational surveys and N-body simulations. In this section, we provide an overview of the available functionality. The set of algorithms currently implemented is not meant to be exhaustive, but instead a solid foundation for LSS data analysis.

3.4.1. *Power Spectra*

For simulation boxes with periodic boundary conditions, the `FFTPower` algorithm measures the power directly from the square of the Fourier modes of the overdensity field. The 1D or 2D power spectrum, $P(k)$ or $P(k,\mu)$, can be computed, as well as the power spectrum multipoles $P_\ell(k)$. Here, μ represents the angle cosine between the pair separation vector and the line-of-sight. For observational data, in the form of right ascension (RA), declination (Dec), and redshift, the power spectrum multipoles of the density field can be computed using the `ConvolvedFFTPower` algorithm. The implementation uses the FFT-based estimator described in Hand et al. (2017a), which requires $2\ell+1$ FFTs to compute a given multipole of order ℓ. This estimator improves the FFT-based estimator presented by Bianchi et al. (2015) and Scoccimarro (2015), building on the ideas of previous power spectrum estimators (Feldman et al. 1994; Yamamoto et al. 2006), and in particular, the treatment of the anisotropic 2PCF using spherical harmonics of Slepian & Eisenstein (2015b). We also provide the `ProjectedFFTPower` for computing the power spectrum of a field in a simulation box, projected along the specified axes. Such an observable is useful for e.g., Lyman-α or weak lensing data analysis. The correctness of these algorithms has been verified using independent implementations from within the Baryon Oscillation Spectroscopic Survey (BOSS) collaboration.

3.4.2. *2-Point Correlation Functions*

`nbodykit` includes functionality for counting pairs of objects and computing their correlation function in configuration space. We leverage the blazing speed of the publicly available `Corrfunc` chaining mesh code for these calculations (Sinha & Garrison 2017). We adapt its highly optimized pair counting routines to perform calculations using MPI. We perform a domain decomposition on the input data such that the objects on a particular MPI rank are spatially confined to include all pairs within the maximum separation. For nonuniform density fields, the domain decomposition results in a particle load that is balanced across MPI

ranks.[14] The relevant pair counting algorithms are `SimulationBoxPairCount` and `SurveyDataPairCount`. These classes can count pairs of objects as a function of the 3D separation r, the separation r and angle to the line-of-sight μ, the angular separation θ, and the projected distances perpendicular r_p and parallel π to the line-of-sight.

Users can compute the correlation function of a Catalog using the `SimulationBox2PCF` and `SurveyData2PCF` classes, which internally rely on the previously described pair counting classes. For data with periodic boundary conditions, we use analytic randoms to estimate the correlation function using the so-called "natural" estimator: $DD/RR - 1$. A Catalog object holding synthetic randoms can be supplied, in which case the Landy-Szalay estimator (Landy & Szalay 1993) is employed: $(DD - 2DR + RR)/RR$. The variations of the correlation function that can be computed by these two classes are as follows:

- as a function of three-dimensional separation, $\xi(r)$
- accounting for the angle to the line-of-sight, $\xi(r,\mu)$ and $\xi(r_p,\pi)$
- as a function of angular separation, $w(\theta)$
- projected over the line-of-sight separations, $w_p(r_p)$

The correctness of the pair counting and correlation function algorithms described here was independently verified using the `kdcount` and `Halotools` software.

3.4.3. *3-Point Correlation Function*

The `SimulationBox3PCF` and `SurveyData3PCF` classes compute the multipoles of the isotropic 3-point correlation function (3PCF) in configuration space. The algorithm follows the implementation described in Slepian & Eisenstein (2015b), which scales as $\mathcal{O}(N^2)$, where N is the number of objects. Their improved estimator relies on a spherical harmonic decomposition to achieve a similar scaling with N as two-point clustering estimators. We note that the FFT-based implementation of this algorithm (presented in Slepian & Eisenstein 2016) and the anisotropic version described in Slepian & Eisenstein (2017) have not yet been implemented, although there are plans to do so in the future. We have verified the accuracy of the isotropic 3PCF classes against the implementation used in Slepian & Eisenstein (2015b). An implementation of this algorithm including anisotropy written in C++ and optimized for HPC machines was recently presented in Friesen et al. (2017).

---

[14] We thank Biwei Dai for the implementation of the load balancer.

3.4.4. *Grouping Methods*

The `FOF` class implements the well-known Friends-of-Friends algorithm, which identifies clusters of points that are spatially less distant than a threshold linking length. It uses a parallel implementation of the algorithm described in Feng & Modi (2017), which utilizes KD-trees and the `kdcount` software. FOF groups can be identified as a function of three-dimensional or angular separation. We also provide functions for transforming the output of the `FOF` algorithm to a Catalog of halo objects (a `HaloCatalog`) in a manner compatible with the `Halotools` software.

`nbodykit` can also identify clusters of objects using a cylindrical rather than spherical geometry. We implement a parallel version of the algorithm described in Okumura et al. (2017) in the `CylindricalGroups` class. Our implementation relies on the neighbor querying capability of `kdcount` and the group-by methods of `pandas`.

Finally, the `FiberCollisions` class simulates the process of assigning spectroscopic fibers to objects in a fiber-fed redshift survey such as BOSS or eBOSS (Dawson et al. 2013, 2016). This procedure results in so-called "fiber collisions" when two objects are separated by an angular width on the sky that is smaller than the fiber size. We follow the procedure outlined in Guo et al. (2012) to assign fibers to an input catalog of objects. We identify angular FOF groups using a linking length equal to the fiber collision scale and assign fibers to the objects in such a manner as to minimize the number of objects that do not receive a fiber.

3.4.5. *Miscellaneous*

`nbodykit` also includes algorithms that generally serve a supporting role in other algorithms. The `KDDensity` class estimates a proxy density quantity for an input set of objects using the inverse cube of the distance to an object's nearest neighbor. The `RedshiftHistogram` class computes the mean number density as a function of redshift, $n(z)$, from an input catalog of objects. We plan to generalize this algorithm to be a more universal histogram calculator that could, for example, compute mass or luminosity functions.

4. DEVELOPMENT WORKFLOW

4.1. *Version Control*

`nbodykit` is developed using the version control features of `git`,[15] and the code is hosted in a public repos-

itory on GitHub.[16] Major changes to the code base are performed using a *pull request* workflow, which provides a mechanism for developers to review changes before they are merged into the main source code. Users can contribute to `nbodykit` by first *forking* the main repository, making changes in this fork and submitting the changes to the main repository via a pull request. This workflow helps assure the overall quality of the code base and ensures that new changes are properly documented and tested. Bugs and new feature requests can be submitted as GitHub issues. Alternatively, users can send an email to nbodykit-issues@fire.fundersclub.com, which will automatically open an issue on GitHub. As `nbodykit` is intended as a community-based resource, we encourage user contributions and ideas for new functionality. We adopt a "mentoring" approach for new features and will gladly offer advice and guidance to new users who wish to contribute to `nbodykit` for the first time.

4.2. *Automated Testing with MPI Support*

`nbodykit` is extensively tested via hundreds of unit tests using the `runtests`[17] package (Feng & Hand 2017). As `mpi4py` does not provide a reusable framework for testing parallel applications, we have developed `runtests` to fill this gap in the development process. It extends the `py.test`[18] testing framework, adding several features. First, the test driver incrementally rebuilds and installs the Python package before running the test suite. Second, it adds MPI support by allowing users to specify the number of processes with which each test function should be executed. It also supports computing the testing coverage for parallel applications, where test coverage is defined as the percentage of the software covered by the test suite.

We execute the `nbodykit` test suite via the continuous integration (CI) service Travis,[19] using `runtests` to test both serial and parallel execution of the code. The test suite is currently executed on both Linux and Mac OS X operating systems and for Python versions 2.7, 3.5, and 3.6. Whenever a pull request is opened, the test suite is executed and the new changes will not be merged if the test suite fails. We also compute the testing coverage of the code base. Currently, `nbodykit` maintains a value of 95%. We use the Coveralls[20] service to ensure that

---

[15] http://git-scm.com

[16] http://github.com/bccp/nbodykit

[17] https://github.com/rainwoodman/runtests

[18] http://pytest.org

[19] https://travis-ci.org

[20] https://coveralls.io

new changes cannot be merged into the main repository if the testing coverage decreases.

4.3. *Use on Personal and HPC Machines*

nbodykit is compatible with both Python versions 2.7 and 3.x. For personal computing systems (Mac OS X and Linux), we provide binaries of nbodykit and its dependencies on the Berkeley Center for Cosmological Physics (BCCP) Anaconda channel.[21] nbodykit (and all of its dependencies) can be installed into an Anaconda environment using a simple command: `conda install -c bccp nbodykit`. We ensure all packages on the BCCP channel are up-to-date using a nightly cron job hosted on the Travis CI service.

Supercomputing systems often require recompiling the dependencies of nbodykit using the machine-specific compilers and MPI configuration. For example, we use the "conda build" functionality of the Anaconda package to compile and update nbodykit and its dependencies nightly on the NERSC Cray supercomputers. The infrastructure for building nbodykit and its dependencies is publicly available on GitHub,[22] which users can re-use to setup nbodykit on HPC machines other than NERSC. However, we recommend that users first test if the default binaries on the BCCP channel are compatible with their supercomputing environment.

The remaining barrier to using nbodykit on HPC systems is the incompatibility of the Python launch system and the shared file systems of HPC machines. When launching an MPI application using Python, the file system will stall when all of the Python instances (can be thousands or more) query the file system for modules on the search path. This issue effectively prevents the use of Python applications on HPC machines.

nbodykit utilizes an open-source solution, denoted "python-mpi-bcast", to facilitate deploying Python applications on HPC machines (Feng & Hand 2016). This tool bundles and delivers runtime dependencies to the HPC computing nodes via an MPI broadcast operation, bypassing the file system bottleneck and allowing Python applications to launch at near-native speed. Users can modify their job scripts in a non-invasive manner to deploy our tool. Additional details and setup instructions can be found in Feng & Hand (2016). The tool is publicly available on GitHub.[23]

4.4. *Documentation*

Documentation for nbodykit is available on Read the Docs.[24] The documentation is generated using Sphinx[25] and includes comprehensive documentation of the nbodykit API. It also includes detailed walkthroughs of each of the main components of nbodykit.

We provide a set of recipes detailing a broad selection of the functionality available in nbodykit in the "Cookbook" section of the documentation. Ranging from simple tasks to more complex work flows, we hope that these recipes help users become acclimated to nbodykit as well as illustrate the power of nbodykit for LSS data analysis. The recipes are in the form of Jupyter notebooks. An interactive environment containing the recipe notebooks is available to users via the Binder service.[26] This allows new users to explore nbodykit without installing nbodykit on their own machine.

5. IN ACTION

In this section, we describe a realistic LSS application using nbodykit: a galaxy clustering emulator. The goal of the emulator is to produce the galaxy power spectrum from first principles, given a background cosmological model. The application combines several components of nbodykit to achieve this goal. The steps include:

- Initial conditions: the LinearMesh class creates a Gaussian realization of a density field in Fourier space from an input power spectrum.

- N-body simulation: the initial conditions are evolved forward to $z = 0$ using the FastPM quasi-N-body particle mesh scheme of Feng et al. (2016).

- Halo Identification: halos are identified from the matter field using the FOF grouping algorithm.

- Halo Population: halos are populated with galaxies using the HOD from Zheng et al. (2007) and the Halotools package.

- Clustering Estimation: $P(k)$ is computed for each of the above steps using the FFTPower algorithm.

We diagram the flow of data and parameters for these steps in the top right panel of Figure 5. We also show the source code for the application using nbodykit, which can be implemented using only ~30 lines of code. We emphasize that with the component-based approach of nbodykit, the user is free to output and serialize any intermediate data products during the execution of the

---

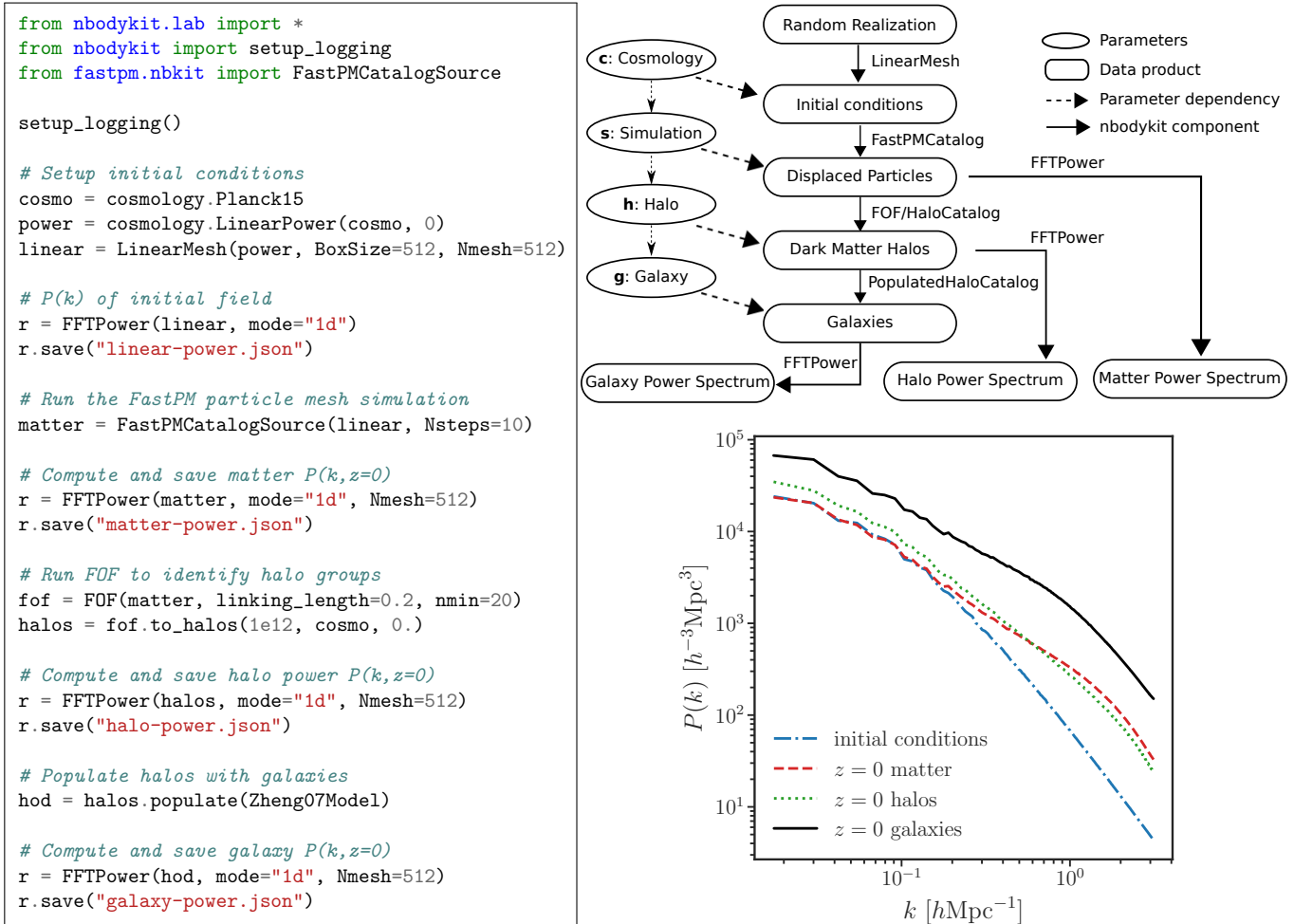[21] https://anaconda.org/bccp
[22] https://github.com/bccp/conda-channel-bccp
[23] https://github.com/rainwoodman/python-mpi-bcast
[24] http://nbodykit.readthedocs.io
[25] http://www.sphinx-doc.org
[26] https://mybinder.org

```python
from nbodykit.lab import *
from nbodykit import setup_logging
from fastpm.nbkit import FastPMCatalogSource

setup_logging()

# Setup initial conditions
cosmo = cosmology.Planck15
power = cosmology.LinearPower(cosmo, 0)
linear = LinearMesh(power, BoxSize=512, Nmesh=512)

# P(k) of initial field
r = FFTPower(linear, mode="1d")
r.save("linear-power.json")

# Run the FastPM particle mesh simulation
matter = FastPMCatalogSource(linear, Nsteps=10)

# Compute and save matter P(k,z=0)
r = FFTPower(matter, mode="1d", Nmesh=512)
r.save("matter-power.json")

# Run FOF to identify halo groups
fof = FOF(matter, linking_length=0.2, nmin=20)
halos = fof.to_halos(1e12, cosmo, 0.)

# Compute and save halo power P(k,z=0)
r = FFTPower(halos, mode="1d", Nmesh=512)
r.save("halo-power.json")

# Populate halos with galaxies
hod = halos.populate(Zheng07Model)

# Compute and save galaxy P(k,z=0)
r = FFTPower(hod, mode="1d", Nmesh=512)
r.save("galaxy-power.json")
```


Figure 5. A galaxy clustering emulator, implemented with `nbodykit`. *Left*: the source code for the application, which evolves an initial Gaussian field to $z = 0$ using the FastPM simulation scheme, identifies FOF halos, populates those halos with galaxies, and records the power spectrum of each step. *Right, top*: the flow of data through the various components. *Right, bottom*: the resulting $P(k)$ measured for each step in the emulator. Performance benchmarks for this application are given in Figure 7.

larger application, as we have done in this example for the power spectra of the initial, matter, and halo density fields. Finally, note that the source code in Figure 5 can be executed with an arbitrary number of MPI ranks. We discuss performance benchmarks for this application as a function of the number of MPI processes in the next section.

6. PERFORMANCE BENCHMARKS

In this section, we present performance benchmarks for several `nbodykit` algorithms, as well as the emulator application discussed in Section 5. Tests are run on the NERSC Cori Phase I Haswell nodes, with 32 MPI cores per node. In Figure 6, we show the strong scaling results for the `FFTPower`, `ConvolvedFFTPower`, `SimulationBoxPairCount`, and `SimulationBox3PCF` algorithms. The benchmarks are performed for two different data configurations, meant to simulate the data

sets of current and future surveys, denoted as "small" and "large", respectively. The "small" sample is modeled after the completed BOSS galaxy sample (Reid et al. 2016) and includes 10^6 galaxies in a cubic box of side length $L = 2500 \ h^{-1}\text{Mpc}$. The "large" sample includes a factor of 10 more objects in a box of side length $L = 5000 \ h^{-1}\text{Mpc}$ and is meant to represent data from future surveys such as DESI (DESI Collaboration et al. 2016). We run four sets of benchmarking tests:

- `FFTPower`: compute $P(k, \mu)$ for 10 μ bins, using a mesh size of $N_{\text{mesh}} = 1024$. This requires a single FFT operation.

- `ConvolvedFFTPower`: compute multipoles $P_\ell(k)$ for $\ell = 0$, 2, and 4 for survey data (RA, Dec, z), using a mesh size of $N_{\text{mesh}} = 1024$. The algorithm requires $2\ell + 1$ FFT operations per multipole, and 15 in total for this test.
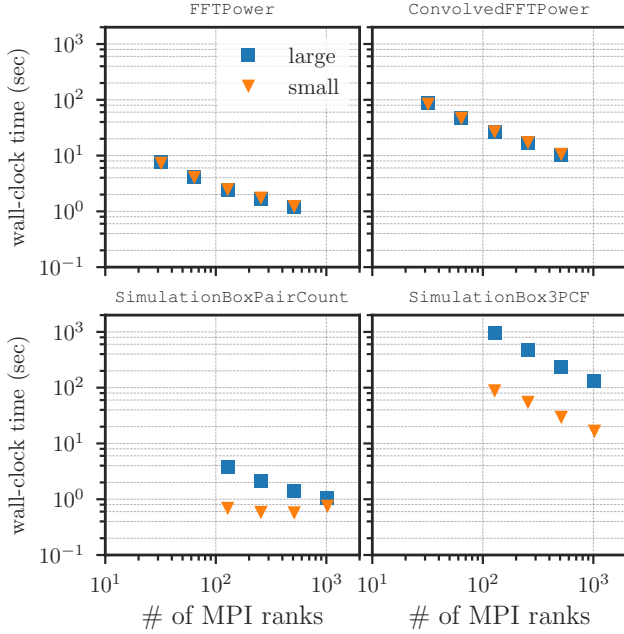
Figure 6. Performance benchmarks for four `nbodykit` algorithms for our "small" data set (10^6 objects) and our "large" data set (10^7 objects). The algorithms in the top row use FFT-based estimators to compute power spectra, while those in the bottom row of panels count pairs of objects in configuration space. The FFT-based algorithms take near-identical time for the large and small data sets due to the use of a 1024^3 mesh in both cases. The benchmarks were performed on the NERSC Cori Phase I Haswell nodes using 32 MPI ranks per node. See the text of Section 6 for further details on the test configurations.

- `SimulationBoxPairCount`: count the number of pairs as a function of separation for 10 separation bins ranging from $r = 10\ h^{-1}$Mpc to $r = 150\ h^{-1}$Mpc and 100 μ bins.

- `SimulationBox3PCF`: compute the isotropic 3PCF multipoles for $\ell = 0, 1, ..., 10$ and 10 separation bins ranging from $r = 10\ h^{-1}$Mpc to $r = 150\ h^{-1}$Mpc.

In general, these four algorithms show excellent strong scaling with the number of MPI ranks. For the power spectrum algorithms (top row of Figure 6), the dominant calculation is the FFT operation, which has good scaling behavior. Because the FFT is the dominant time cost, we find nearly identical performances for the "small" and "large" samples. The wall-clock time for the `ConvolvedFFTPower` algorithm is roughly fifteen times that of the `FFTPower` algorithm, which is driven by the total number of FFTs that each algorithm computes. The pair-counting-based algorithms both take $\mathcal{O}(N^2)$ time to compute their results. However, the `SimulationBoxPairCount` algorithm relies on
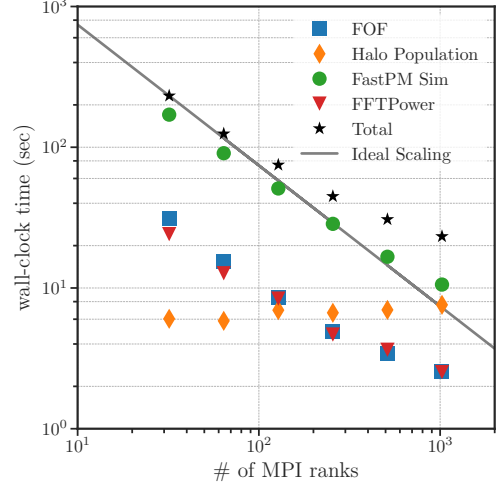
Figure 7. The wall-clock time as a function of the number of MPI ranks used for each step in the galaxy clustering emulator detailed in Figure 5. Overall, the application shows excellent scaling behavior, with deviations from the ideal scaling caused by the halo population step. This step does not currently have a massively parallel implementation and takes a roughly constant amount of time as more cores are used. The benchmarks were performed on the NERSC Cori Phase I Haswell nodes using 32 MPI ranks per node.

the highly optimized `Corrfunc` software, which is significantly faster than `SimulationBox3PCF`, which relies on `kdcount`. When using `SimulationBoxPairCount` on the "small" data set, we find that MPI communication costs are non-negligible due to the relatively small sample size, which hinders the scaling performance of the code.

We also present performance benchmarks for the emulator application described in Section 5. For this test, we run a FastPM particle mesh simulation with 512^3 total particles. The halo finder identifies roughly 225,000 dark matter halos that are then used to build a mock galaxy catalog. The wall-clock times for each step in the emulator are shown in Figure 7. We see that the dominant fraction of the wall-clock time is spent in the FastPM step, which shows excellent strong scaling behavior up to the number of cores we have tested. The implementation of the halo population step using `Halotools` is not massively parallel, and therefore, the time to solution for this step remains relatively constant as the number of cores increases. The wall-clock time for this step only becomes significant as the number of cores approaches ~ 1024, and it would be worth investigating improving this step's scaling if users wish to run often at this scale. However, in our experience, we have not found that the time cost of this step justifies further efforts converting it to a massively parallel implementation.

We emphasize that for all benchmarks presented in this section, the number of MPI ranks can always be increased such that the time to solution is on the order of seconds. This becomes important for realistic data analyses in LSS, which often require repeating an algorithm's calculation hundreds to thousands of times, e.g., while sampling a parameter space using Markov Chain Monte Carlo or optimization techniques. Due to the availability of large-scale computing resources and the scaling behavior of `nbodykit` demonstrated here, we believe that `nbodykit` will be able to meet the computational demands of future LSS data analyses.

7. CONCLUSIONS

We have presented the first public release of `nbodykit` (v0.3.0), a massively parallel Python toolkit for the analysis of large-scale structure data. Relying on the `mpi4py` binding of MPI, the package includes parallel implementations of a set of canonical algorithms in the field of large-scale structure, including two and three-point clustering estimators, halo identification and population tools, and quasi-N-body simulation schemes. The toolkit also includes a set of distributed data containers that support a variety of data formats common in astronomy, including CSV, FITS, HDF5, binary, and `bigfile` data. With these tools, we hope `nbodykit` can serve as a foundation for the community to build upon as we strive to meet the demands of future LSS data sets.

In designing `nbodykit`, we have attempted to balance the requirements of both a scalable and interactive piece of software. Our ultimate goal was to produce a piece of software that is as usable in a Jupyter notebook environment as on an HPC machine. We have adopted a modular, component-based approach that should enable researchers to integrate `nbodykit` with their own software to build complicated applications. As an illustration of its power, we have discussed an implementation of a galaxy clustering emulator using `nbodykit`, which provides a complete forward model for the galaxy power spectrum, starting from an initial, Gaussian density field. We have also demonstrated that the toolkit shows excellent scaling behavior, presenting a set of performance benchmarks for the emulator as well as some of the more commonly used algorithms in `nbodykit`.

We have outlined our development workflow for producing a piece of reusable scientific software. `nbodykit` is open-source—we strongly believe in the idea of open science and have placed an emphasis on reproducibility when designing `nbodykit`. Designed for the LSS community, we hope that new users will find `nbodykit` useful in their own research and that the software can continue to grow and mature in new ways from community feedback and contributions. We are also strong believers in the necessity of unit testing and adequate documentation for open-source tools. We have attempted to meet these goals using the Travis automated testing service and the Read the Docs documentation hosting tools. Finally, we have aimed to make `nbodykit` widely available and easily installable. The package supports both Python versions 2 and 3, and binary distributions of `nbodykit` and its dependencies can be installed onto Mac OS X and Linux machines using the Anaconda package manager.

In the future, we hope to incorporate the expertise of new developers, from both the LSS and Python HPC communities. We expect the knowledge of both communities to be necessary in the data analysis of future surveys. The set of features currently implemented in `nbodykit` is not meant to be all-inclusive but rather a sampling of the more commonly used tools in the field. Most importantly, we hope that `nbodykit` provides a solid basis for the community to build upon. We warmly welcome feedback and contributions of all forms from the community. As open-source software, `nbodykit` was designed as a tool to help the LSS community, and we hope that community contributions can help maximize its benefits for its users.

REFERENCES

Agrawal, A., Makiya, R., Chiang, C.-T., et al. 2017, JCAP, 10, 003

Alam, S., Ata, M., Bailey, S., et al. 2017, MNRAS, 470, 2617

Astropy Collaboration, Robitaille, T. P., Tollerud, E. J., et al. 2013, A&A, 558, A33

Behroozi, P. S., Wechsler, R. H., & Wu, H.-Y. 2013, ApJ, 762, 109

Beutler, F., Saito, S., Brownstein, J. R., et al. 2014, MNRAS, 444, 3501

Bianchi, D., Gil-Marín, H., Ruggeri, R., & Percival, W. J. 2015, MNRAS, 453, L11

Blake, C., & Glazebrook, K. 2003, ApJ, 594, 665

Blas, D., Lesgourgues, J., & Tram, T. 2011, JCAP, 7, 034

Castorina, E., & White, M. 2017, ArXiv e-prints, arXiv:1709.09730

Cole, S., Fisher, K. B., & Weinberg, D. H. 1995, MNRAS, 275, 515

Cole, S., Percival, W. J., Peacock, J. A., et al. 2005, MNRAS, 362, 505

Coles, P., & Jones, B. 1991, MNRAS, 248, 1

Collette, A., & contributors. 2017, HDF5 for Python, , . http://www.h5py.org

Cui, W., Liu, L., Yang, X., et al. 2008, ApJ, 687, 738

Dalcín, L., Paz, R., Storti, M., & DElía, J. 2008, Journal of Parallel and Distributed Computing, 5, 655

Dalcin, L. D., Paz, R. R., Kler, P. A., & Cosimo, A. 2011, Advances in Water Resources, 34, 1124 , new Computational Methods and Software Tools. http://www.sciencedirect.com/science/article/pii/S0309170811000777

Dask Development Team. 2016, Dask: Library for dynamic task scheduling, , . http://dask.pydata.org

Daubechies, I., ed. 1992, Ten lectures on wavelets

Davis, M., Efstathiou, G., Frenk, C. S., & White, S. D. M. 1985, ApJ, 292, 371

Dawson, K. S., Schlegel, D. J., Ahn, C. P., et al. 2013, AJ, 145, 10

Dawson, K. S., Kneib, J.-P., Percival, W. J., et al. 2016, AJ, 151, 44

Delubac, T., Bautista, J. E., Busca, N. G., et al. 2015, A&A, 574, A59

DESI Collaboration, Aghamousa, A., Aguilar, J., et al. 2016, ArXiv e-prints, arXiv:1611.00036

Desjacques, V., & Seljak, U. 2010, Classical and Quantum Gravity, 27, 124011

Diemer, B. 2017, ArXiv e-prints, arXiv:1712.04512

Ding, Z., Seo, H.-J., Vlah, Z., et al. 2017, ArXiv e-prints, arXiv:1708.01297

Efstathiou, G., Sutherland, W. J., & Maddox, S. J. 1990, Nature, 348, 705

Eisenstein, D. J., & Hu, W. 1999, ApJ, 511, 5

Eisenstein, D. J., Hu, W., & Tegmark, M. 1998, ApJL, 504, L57

Eisenstein, D. J., Zehavi, I., Hogg, D. W., et al. 2005, ApJ, 633, 560

Feldman, H. A., Kaiser, N., & Peacock, J. A. 1994, ApJ, 426, 23

Feng, Y. 2017a, pfft-python, v0.1.13, Zenodo, doi:10.5281/zenodo.1051308. https://doi.org/10.5281/zenodo.1051308

—. 2017b, pmesh, v0.1.33, Zenodo, doi:10.5281/zenodo.1051254. https://doi.org/10.5281/zenodo.1051254

—. 2017c, bigfile, v0.1.39, Zenodo, doi:10.5281/zenodo.1051252. https://doi.org/10.5281/zenodo.1051252

—. 2017d, kdcount, v0.2.9, Zenodo, doi:10.5281/zenodo.1051244. https://doi.org/10.5281/zenodo.1051244

—. 2017e, fastpm-python, v0.0.6, Zenodo, doi:10.5281/zenodo.1051310.
https://doi.org/10.5281/zenodo.1051310

Feng, Y., Chu, M.-Y., Seljak, U., & McDonald, P. 2016, MNRAS, 463, 2273

Feng, Y., & Hand, N. 2016, in Proceedings of the 15th Python in Science Conference, ed. Sebastian Benthall & Scott Rostrup, 137 – 143

Feng, Y., & Hand, N. 2017, runtests, v0.0.23, Zenodo, doi:10.5281/zenodo.1051306.
https://doi.org/10.5281/zenodo.1051306

Feng, Y., & Modi, C. 2017, Astronomy and Computing, 20, 44

Font-Ribera, A., Kirkby, D., Busca, N., et al. 2014, JCAP, 5, 027

Friesen, B., Patwary, M. M. A., Austin, B., et al. 2017, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17 (New York, NY, USA: ACM), 20:1–20:11.
http://doi.acm.org/10.1145/3126908.3126927

Guo, H., Zehavi, I., & Zheng, Z. 2012, ApJ, 756, 127

Hamilton, A. J. S. 2000, MNRAS, 312, 257

Hand, N., & Feng, Y. 2017, classylss, v0.2.7, Zenodo, doi:10.5281/zenodo.1051256.
https://doi.org/10.5281/zenodo.1051256

Hand, N., Li, Y., Slepian, Z., & Seljak, U. 2017a, JCAP, 7, 002

Hand, N., Seljak, U., Beutler, F., & Vlah, Z. 2017b, JCAP, 10, 009

Hearin, A. P., Zentner, A. R., van den Bosch, F. C., Campbell, D., & Tollerud, E. 2016, MNRAS, 460, 2552

Hearin, A. P., Campbell, D., Tollerud, E., et al. 2017, AJ, 154, 190

Hinshaw, G., Larson, D., Komatsu, E., et al. 2013, ApJS, 208, 19

Hockney, R. W., & Eastwood, J. W. 1981, Computer Simulation Using Particles

Jones, E., Oliphant, T., Peterson, P., et al. 2001–2017, SciPy: Open source scientific tools for Python, , .
http://www.scipy.org/

Komatsu, E., Dunkley, J., Nolta, M. R., et al. 2009, ApJS, 180, 330

Komatsu, E., Smith, K. M., Dunkley, J., et al. 2011, ApJS, 192, 18

Krauss, L. M., & Turner, M. S. 1995, General Relativity and Gravitation, 27, 1137

Landy, S. D., & Szalay, A. S. 1993, ApJ, 412, 64

Leauthaud, A., Tinker, J., Behroozi, P. S., Busha, M. T., & Wechsler, R. H. 2011, ApJ, 738, 45

Lesgourgues, J. 2011, ArXiv e-prints, arXiv:1104.2932

Lesgourgues, J., & Pastor, S. 2006, PhR, 429, 307

Lewis, A., Challinor, A., & Lasenby, A. 2000, ApJ, 538, 473

Maddox, S. J., Efstathiou, G., Sutherland, W. J., & Loveday, J. 1990, MNRAS, 242, 43P

McKinney, W. 2010, in Proceedings of the 9th Python in Science Conference, ed. S. van der Walt & J. Millman, 51 – 56

Merz, H., Pen, U.-L., & Trac, H. 2005, NewA, 10, 393

Modi, C., Castorina, E., & Seljak, U. 2016, ArXiv e-prints, arXiv:1612.01621

Momcheva, I., & Tollerud, E. 2015, ArXiv e-prints, arXiv:1507.03989

Moore, A. W., Connolly, A. J., Genovese, C., et al. 2001, in Mining the Sky, ed. A. J. Banday, S. Zaroubi, & M. Bartelmann, 71

Mueller, E.-M., Percival, W., Linder, E., et al. 2016, ArXiv e-prints, arXiv:1612.00812

NSF. 2017, NSF Committee on Software Infrastructure for Heterogeneous Computing, , .
https://github.com/labarba/NSFcommittee-SI2017/

Okumura, T., Takada, M., More, S., & Masaki, S. 2017, MNRAS, 469, 459

Ostriker, J. P., & Steinhardt, P. J. 1995, Nature, 377, 600

Perez, F., & Granger, B. E. 2007, Computing in Science Engineering, 9, 21

Perlmutter, S., Aldering, G., Goldhaber, G., et al. 1999, ApJ, 517, 565

Pinol, L., Cahn, R. N., Hand, N., Seljak, U., & White, M. 2017, JCAP, 4, 008

Pippig, M. 2013, SIAM Journal on Scientific Computing, 35, C213

Planck Collaboration, Ade, P. A. R., Aghanim, N., et al. 2014, A&A, 571, A16

—. 2016, A&A, 594, A13

Reid, B., Ho, S., Padmanabhan, N., et al. 2016, MNRAS, 455, 1553

Riess, A. G., Filippenko, A. V., Challis, P., et al. 1998, AJ, 116, 1009

Schmittfull, M., Baldauf, T., & Zaldarriaga, M. 2017, PhRvD, 96, 023505

Scoccimarro, R. 2015, PhRvD, 92, 083532

Sefusatti, E., Crocce, M., Scoccimarro, R., & Couchman, H. M. P. 2016, MNRAS, 460, 3624

Seo, H.-J., & Eisenstein, D. J. 2003, ApJ, 598, 720

Sheldon, E. 2017, A python package for FITS input/output wrapping cfitsio, , .
https://github.com/esheldon/fitsio

Sinha, M., & Garrison, L. 2017, Corrfunc: Blazing fast correlation functions on the CPU, Astrophysics Source Code Library, , , ascl:1703.003

Slepian, Z., & Eisenstein, D. J. 2015a, ArXiv e-prints, arXiv:1510.04809

—. 2015b, MNRAS, 454, 4142

—. 2016, MNRAS, 455, L31

—. 2017, ArXiv e-prints, arXiv:1709.10150

Slepian, Z., Eisenstein, D. J., Brownstein, J. R., et al. 2017, MNRAS, 469, 1738

Slosar, A., Hirata, C., Seljak, U., Ho, S., & Padmanabhan, N. 2008, JCAP, 8, 031

Smith, R. E., Peacock, J. A., Jenkins, A., et al. 2003, MNRAS, 341, 1311

Springel, V. 2005, MNRAS, 364, 1105

Springel, V., Yoshida, N., & White, S. D. M. 2001, NewA, 6, 79

Takahashi, R., Sato, M., Nishimichi, T., Taruya, A., & Oguri, M. 2012, ApJ, 761, 152

Tassev, S., Zaldarriaga, M., & Eisenstein, D. J. 2013, JCAP, 6, 036

Thomas, K., Ragan-Kelley, B., Pérez, F., et al. 2016, in Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, ELPUB

Turk, M. J., Smith, B. D., Oishi, J. S., et al. 2011, ApJS, 192, 9

van der Walt, S., Colbert, S. C., & Varoquaux, G. 2011, Computing in Science and Engineering, 13, 22

Vlah, Z., Seljak, U., & Baldauf, T. 2015, PhRvD, 91, 023508

Waters, D., Di Matteo, T., Feng, Y., Wilkins, S. M., & Croft, R. A. C. 2016, MNRAS, 463, 3520

White, M. 2002, ApJS, 143, 241

White, M., Tinker, J. L., & McBride, C. K. 2014, MNRAS, 437, 2594

Yamamoto, K., Nakamichi, M., Kamino, A., Bassett, B. A., & Nishioka, H. 2006, PASJ, 58, 93

Yang, Y.-B., Feng, L.-L., Pan, J., & Yang, X.-H. 2009, Research in Astronomy and Astrophysics, 9, 227

Zheng, Z., Coil, A. L., & Zehavi, I. 2007, ApJ, 667, 760