Mon. Not. R. Astron. Soc. **000**, ??–?? () Printed 19 May 2016 (MN LATEX style file v2.2)

High-Precision Modeling of Large-Scale Structure: An open source approach with Halotools

Andrew P. Hearin[1], Duncan Campbell[2], Erik Tollerud[2,3]
many others
[1] *Yale Center for Astronomy & Astrophysics, Yale University, New Haven, CT*
[2] *Department of Astronomy, Yale University, P.O. Box 208101, New Haven, CT*
[3] *Space Telescope Science Institute, Baltimore, MD 21218, USA*

19 May 2016

ABSTRACT

We present the first official release of Halotools (v0.2), a community-driven python package designed to build and test models of the galaxy-halo connection. Halotools provides a modular platform for creating mock universes with a rich variety of models of galaxy evolution, such as the HOD, CLF, abundance matching, assembly biased models, cored/cuspy NFW profiles, velocity bias, and many other model styles and features. The package has an extensive, heavily optimized toolkit to make mock observations on a synthetic galaxy population, including galaxy clustering, galaxy-galaxy lensing, galaxy group identification, RSD multipoles, void statistics, pairwise velocities and others. Halotools is written in a object-oriented style that enables complex models to be built from a set of simple, interchangeable components, including those of your own creation. Halotools has a rigorously maintained automated testing suite and is exhaustively documented on halotools.readthedocs.org, which includes quickstart guides, source code notes and a large collection of worked examples. The documentation effectively serves as an online textbook on how to build empirical models of galaxy formation with python. We conclude this paper by describing how Halotools can be used to analyze existing datasets to obtain robust constraints on star-formation and quenching processes at low- and high-redshift, and by outlining the Halotools program to help prepare for the arrival of Stage IV dark energy experiments.

1 INTRODUCTION

Halotools is an affiliated package[1] of Astropy (Astropy Collaboration et al. 2013).

2 PACKAGE OVERVIEW

2.1 Optimization strategy

Bounds-checking, exception-handling, and high-level control flow is written in pure python. All algorithms are intentionally formulated to rely on vectorized functions in `Numpy` such as `np.searchsorted, np.unique` and `np.repeat`. Whenever possible, the function is written to be trivially parallelized using python's native `multiprocessing` module.

In many cases there is simply no memory efficient way to vectorize the problem, and it becomes necessary to write explicit loops over large numbers of points. In such situations, care is taken to pinpoint the specific part of the calculation that is the bottleneck; that section, and that section only, is written in cython.

3 ORGANIZATION INTO SUB-PACKAGES

3.1 Empirical Models

3.2 Mock Observations

In the analysis of halo and (mock) galaxy catalogs, many of the same calculations are performed over and over again. How many pairs of points are separated by some distance r? What is the two-point correlation function of some sample of points? What is the host halo mass of some sample of subhalos? What is the local environmental density of some collection of galaxies? It is common to calculate the answers to these and other similar questions in an MCMC-type analysis, when high-performance is paramount. Even outside of the context of likelihood analyses, the sheer size of present-day cosmological simulations presents a formidable computational challenge to

[1] `http://www.astropy.org/affiliated`

© RAS

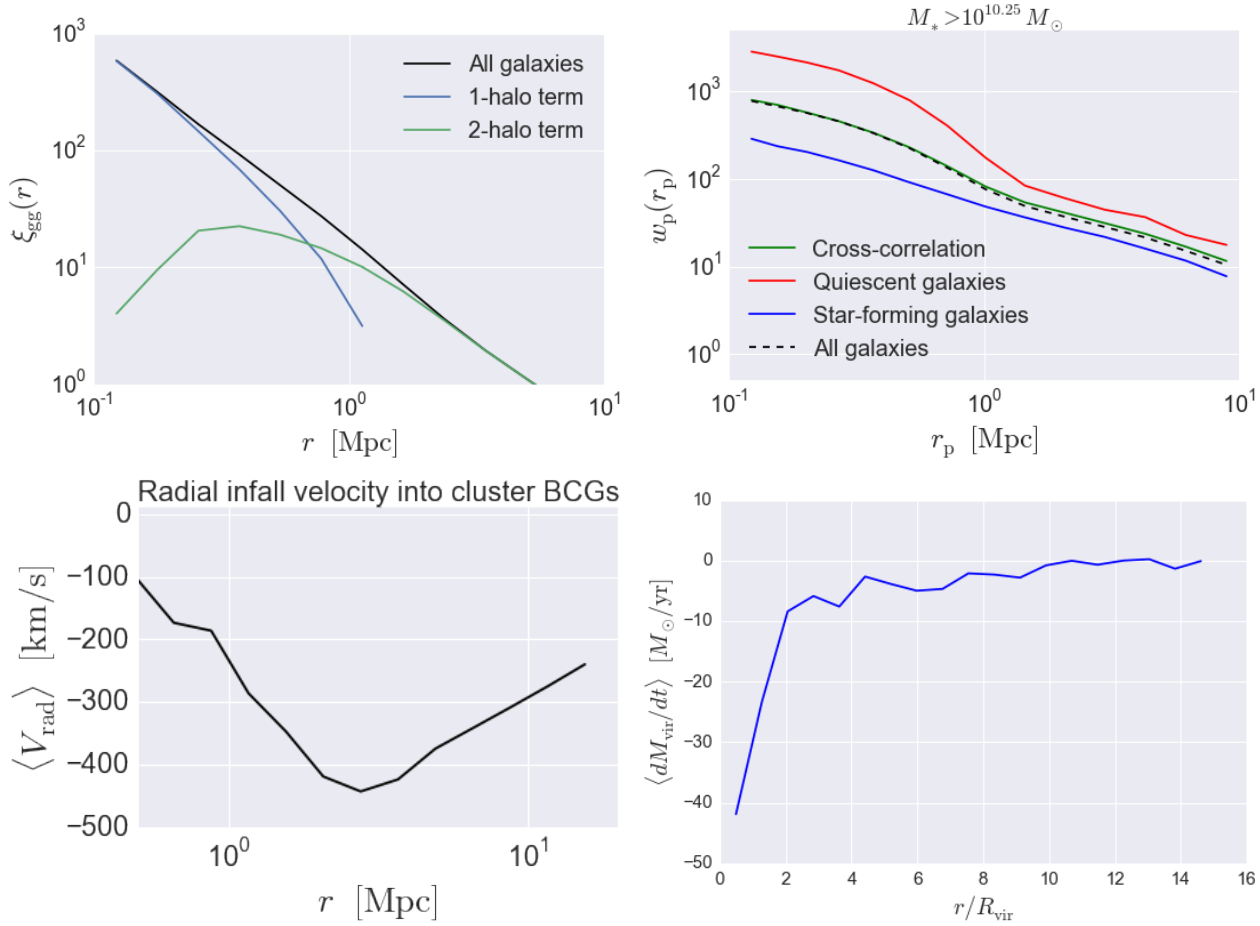

Figure 1. Four example calculations done with Halotools demonstrating the diversity of the `mock_observables` sub-package. Each is part of a tutorial found in `http://halotools.readthedocs.io`, to which we refer the reader for details. Here we only point out that each panel demonstrates the result of a heavily optimized function with a user-friendly API requiring minimal setup. *Top left:* Three-dimensional correlation function of mock galaxies $\xi_{\rm gg}(r)$ split into contributions from pairs of galaxies occupying a common halo (1-halo term), and pairs in distinct halos (2-halo term). *Top right:* Projected correlation function $w_{\rm p}(r_{\rm p})$ of star-forming and quiescent galaxies, as well as their cross-correlation. *Bottom left:* Mean pairwise radial velocity of galaxies in the neighborhood of a cluster BCG. *Bottom right:* As a function of the $R_{\rm vir}$−normalized distance to a cluster, we show the mean mass accretion rate of nearby lower-mass subhalos.

evaluate these functions in a reasonable runtime. There is also the notorious complicating nuisance of properly accounting for the periodic boundary conditions of a simulation. Much research time has been wasted by many different researchers writing their own private versions of these calculations, writing code that is not portable as it was developed making hard assumptions that are only applicable to the immediate problem at hand.

The `mock_observables` sub-package is designed to remedy this situation. This sub-package contains a large collection of functions that are commonly encountered when analyzing halo and galaxy catalogs, including:

- The many variations of two-point correlation functions,
  - three-dimensional correlation function $\xi(r)$,
  - redshift-space correlation function $\xi(r_{\rm p}, \pi)$,
  - projected correlation function $w_{\rm p}(r_{\rm p})$,
  - projected surface density $\Delta\Sigma(r_{\rm p})$ (aka galaxy-galaxy lensing),
  - RSD multipoles $\xi_\ell(s)$.
- marked correlation functions $\mathcal{M}(r)$,
- friends-of-friends group identification,
- *group aggregation* calculations, e.g., calculating the total stellar mass of galaxies of a common group $M_*^{\rm tot}$,
- *isolation criteria*, e.g., identifying those galaxies with no more massive companion inside some search radius,
- pairwise velocity statistics, e.g, the line-of-sight velocity dispersion as a function of projected distance $\sigma_{\rm los}(r_{\rm p})$,
- void probability function $P_{\rm void}(r)$.

The `mock_observables` sub-package contains heavily optimized implementations of all the above functions, as well as a variety of others. Every function in `mock_observables` has a stable, user-friendly API that is consistently applied across the package. The docstring

of all functions contains an explicit example of how to call the function, and in many cases there is a step-by-step tutorial showing how the function might be used in a typical analysis. Considerable effort has been taken to write `mock_observables` to be modular, so that users can easily borrow the algorithm patterns to write their own variation on the provided calculations.

3.3 Managing Simulation Data

4 PACKAGE DEVELOPMENT

4.1 GitHub workflow

Halotools has been developed fully in the open since the inception of the project. Version control for the code base is managed using git[2], and the public version of the code is hosted on GitHub[3]. The latest stable version of the code can be installed via `pip install halotools`, but at any given time the `master` branch of the code on `https://github.com/astropy/halotools` may have features and performance enhancements that are being prepared for the next release. A concerted effort is made to ensure that only thoroughly tested and documented code appears in the public `master` branch, though Halotools users should be aware of the distinction between the bleeding edge version in `master` and the official release version available through `pip`.

Development of the code is managed with a *Fork & Pull* workflow. Briefly, code development begins by creating a private *fork* of the main repository on GitHub. Developers then work only on the code in their fork. In order to incorporate a change to the main repository, it is necessary to issue a *Pull Request* to the `master` branch. The version of the code in the Pull Request is then reviewed by the Halotools developers before it is either rejected or merged into `master`.

4.2 Automated testing

Halotools includes hundreds of unit-tests that are incorporated into the package via the `py.test` framework.[4] These tests are typically small blocks of code that test a specific feature of a specific function. The purpose of the testing framework is both to verify scientific correctness and also to enforce that the API of the package remains stable. We also use *continuous integration*, a term referred to the automated process of running the entire test suite in a variety of different system configurations (e.g., with different releases of `Numpy` and `Astropy` installed, or different versions of the Python language). Each time any Pull Request is submitted to the `master` branch of the code,the proposed new version of the code is copied to a variety of virtual environments, and the entire test suite is run repeatedly in each environment configuration. The Pull Request will not be merged into `master` unless the entire test suite passes in all environment configurations.

[2] `http://git-scm.com`
[3] `http://www.github.com`
[4] `http://pytest.org`

We use `Travis`[5] for continuous integration in Unix environments such as Linux and Mac OS X and `AppVeyor`[6] for Windows environments.

Pull Requests to the `master` branch are additionally subject to a requirement enforced by `Coveralls`.[7] This service performs a static analysis on the Halotools code base and determines the portions of the code that are covered by the test suite, making it straightforward to identify logical branches whose behavior remains to be tested. `Coveralls` issues a report for the fraction of the code base that is covered by the test suite; if the returned value of this fraction is smaller than the coverage fraction of the current version of `master`, the Pull Request is not accepted. This ensures that test coverage can only improve as the code evolves and new features are added.

Any time a bug is found in the code, either by Halotools developers or users, a GitHub Issue is raised calling public attention to the problem. When the Halotools developers have resolved the problem, a corresponding *regression test* becomes a permanent contribution to the code base. The regression test explicitly demonstrates the specific source of the problem, and contains a hyperlink to the corresponding GitHub Issue. The test will fail when executed from the version of the code that had the problem, and will pass in the version with the fix. Regression testing helps makes it transparent how the bug was resolved and protects against the same bug from creeping back into the repository as the code evolves.

4.3 Documentation

Documentation of the code base is generated via sphinx[8] and is hosted on ReadTheDocs[9] at `http://halotools.readthedocs.io`. The public repository `https://github.com/astropy/halotools` has a webhook set up so that whenever there is a change to the `master` branch, the documentation is automatically rebuilt to reflect the most up-to-date version of `master`.

Every user-facing class, method and function in Halotools has a docstring describing its general purpose, its inputs and output, and also providing an explicit example usage. The docstring for many functions with complex behavior comes with a hyperlink to a separate section of the documentation in which mathematical derivations and algorithm notes are provided. The documentation also includes a large number of step-by-step tutorials and example analyses. The goal of these tutorials is more than simple code demonstration: the tutorials are intended to be a pedagogical tool illustrating how to analyze simulations and study models of the galaxy-halo connection in an efficient and reproducible manner.

[5] `https://travis-ci.org`
[6] `https://www.appveyor.com`
[7] `https://coveralls.io`
[8] `http://www.sphinx-doc.org`
[9] `https://readthedocs.io`

REFERENCES

Astropy Collaboration Robitaille T. P., Tollerud E. J., Greenfield P., Droettboom M., Bray E., Aldcroft T., et al., 2013, AAP, 558, A33