Mon. Not. R. Astron. Soc. **000**, ??–?? () Printed 23 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*

23 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, halo 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 constrain galaxy formation physics, 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

Halotools is composed almost entirely in python with syntax that is compatible with both 2.7.x and 3.x versions of the language. Bounds-checking, exception-handling, and high-level control flow are always written in pure python. Whenever possible, performance-critical functions are written to be trivially parallelized using python's native `multiprocessing` module. Halotools relies heavily on vectorized functions in `Numpy` as a core optimization strategy. However, in many cases there is simply no memory efficient way to vectorize a calculation, 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.

Halotools is designed with a high degree of modularity, so that users can pick and choose the features that are suitable to their science applications. At the highest level, this modularity is reflected in the organization of the package into sub-packages. For Halotools v0.2, there are three major sub-packages. The `empirical_models` sub-package described in §2.2 contains the implementation of models of the galaxy-halo connection, as well as a flexible object-oriented platform for users to design their own models. The `mock_observables` sub-package described in §2.3 contains a collection of functions that can be used to generate predictions for models in a manner that can be directly compared to astronomical observations. Many of the functions in `mock_observables` should also be of general use in the analysis of halo catalogs. The `sim_manager` sub-package described in §2.1 is responsible for reducing "raw" halo catalogs into efficiently organized fast-loading hdf5 files, and for creating and keeping track of a persistent memory of where the simulation data is stored on disk.

Although these sub-packages are designed to work

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

together, each individual sub-package has entirely standalone functionality that is intended to be useful even in the absence of the others. For example, while Halotools does not provide pre-processed halo catalogs from the Millennium simulation, the `sim_manager` sub-package can nonetheless be used to process and cache Millennium halos into a convenient, self-expressive format that users of that simulation may find useful. The `empirical_models` sub-package can be used to populate mock galaxies into the the halos of any cosmological simulation, where the populated halos could be identified by any algorithm. The functions in the `mock_observables` sub-package simply accept point-data as inputs, and so these functions could be used to generate observational predictions for semi-analytical models that otherwise have no connection to Halotools.

In the subsections below we outline each of these sub-packages in turn, though we refer the reader to `halotools.readthedocs.io` for more comprehensive descriptions.

2.1 Managing Simulation Data

One of the most tedious tasks in simulation analysis is the initial process of getting started with a halo catalog: reading large data files storing the halos (typically ASCII-formatted), making cuts on halos, adding additional columns, and storing the reduced and value-added catalog on disk for later use. In our experience with simulation analysis, one of the most common sources of bugs comes from these initial bookkeeping exercises.

The `sim_manager` sub-package has been written to standardize this process so that users can get started with full-fledge halo catalog analysis from just a few lines of code. The `sim_manager.RockstarHlistReader` class allows users to quickly create a Halotools-formatted catalog starting from the typical ASCII output of the `Rockstar` halo-finder (Behroozi et al. 2013, 2011). Users wishing to work with catalogs of halos identified by algorithms other than `Rockstar` can use the `sim_manager.TabularAsciiReader` class to initially process their ascii data. Both readers are built around a convenient API that uses Python's native "lazy evaluation" functionality to select on-the-fly only those columns and rows that are of interest, making these readers highly memory efficient.

All Halotools-formatted catalogs are python objects storing the halo catalog itself in the form of an Astropy Table, and also storing some metadata about the simulated halos. In order to build an instance of a Halotools-formatted catalog, a large collection of self-consistency checks about the halo data and metadata are performed, and an exception is raised if any inconsistency is detected. These checks are automatically carried out at the initial processing stage, and also every time the catalog is loaded into memory, to help ensure that the catalog is processed correctly and does not become corrupt over time.

The `sim_manager` sub-package allows users to cache their processed halo catalogs when they are saved to disk, creating the option to load their catalogs into memory with the simple and intuitive syntax shown in code block 1.

The cached simulations are stored in the form of an hdf5 file.[2]. This binary file format is fast-loading and permits metadata to be bound directly to the file in a transparent manner, so that the cached binary file is a self-expressive object. The `sim_manager.HaloTableCache` class provides an object-oriented interface for managing the cache of simulations, but users are also free to work directly with the cache log, which is a simple, human-readable text file stored in $HOME/.astropy/cache/halotools/halo_table_cache_log.txt.

Using the Halotools caching system is optional in every respect. Users who prefer their own system for managing simulated data are free to do so in whatever manner they wish; they need only pass the necessary halo data and metadata to the `sim_manager.UserSuppliedHaloCatalog` class, and the full functionality of all sub-packages of Halotools works with the resulting object instance.

The Halotools developers manage a collection of pre-processed halo catalogs that are available for download either with the `sim_manager.DownloadManager` class, or with the command-line script `halotools/scripts/download_additional_halocat.py`. Through either download method, the catalogs are automatically cached and science-ready as soon as the download completes. Halotools currently offers pre-processed halo catalogs for Rockstar-identified halos from four different simulations: `bolshoi, bolshoi-planck, consuelo` and `multidark`, for which snapshots at $z = 0, 0.5, 1$ and 2 are available.

2.2 Empirical Models

All Halotools models of the galaxy-halo connection are contained in the `empirical_models` sub-package. Halotools models come into two categories: *composite models* and *component models*. A *composite model* is a complete description of the mapping(s) between dark matter halos and all properties of their resident galaxy population. A composite model provides sufficient information to populate an ensemble of halos with a Monte Carlo realization of a galaxy population. All composite models are built from a collection of independently-defined *component models*. A component model provides a map between dark matter halos and a single property of the resident galaxy population. Examples component models include the stellar-to-halo mass relation, an NFW radial profile or the halo mass-dependence of the quenched fraction.

2.2.1 Model styles

Halotools composite models come in two different types: HOD-style models and subhalo-based models. In HOD-style models, there is no connection between the abundance of satellite galaxies in a host halo and the number of subhalos in that host halo. In these models, satellite abundance in each halo is determined by a Monte Carlo

[2] `http://www.h5py.org`

Listing 1: Loading a cached halo catalog into memory

```
from halotools.sim_manager import CachedHaloCatalog
halocat = CachedHaloCatalog(simname = 'bolshoi', redshift = 0.5)

print(halocat.halo_table[0:9]) # view the first ten halos in the catalog
print(halocat.Lbox, halocat.particle_mass) # inspect the halo catalog metadata
```

realization of some analytical model. Examples of this approach to the galaxy-halo connection include the HOD (Berlind & Weinberg 2002) and CLF (Yang et al. 2003), as well as extensions of these that include additional features such as color-dependence (Tinker et al. 2013).

By contrast, in subhalo-based models there is a one-to-one correspondence between subhalos and satellite galaxies. In these models, each host halo in the simulation is connected to a single central galaxy, and each subhalo is connected to a single satellite. Examples include traditional abundance matching (Kravtsov et al. 2004; Conroy et al. 2006), age matching (Hearin & Watson 2013), and parameterized stellar-to-halo mass models (Behroozi et al. 2010; Moster et al. 2010).

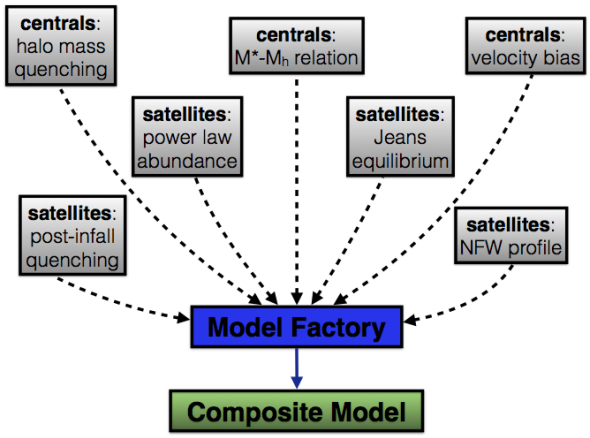

Figure 1. Cartoon example of how an HOD-style model can be built with the Halotools factory design pattern. Users select a set of *component model* features of their choosing, and compose them together into a *composite model* using the appropriate factory. Users wishing to quickly get up-and-running can instead select a prebuilt composite model that ships with the package. Either way, `halotools.readthedocs.io` contains extensive step-by-step tutorials and worked examples demonstrating how to use Halotools to model the galaxy-halo connection.

2.2.2 Prebuilt models

Halotools ships with a handful of fully formed pre-built composite models, each of which has been designed around a model chosen from the literature. All pre-built models can directly populate a simulation with a mock catalog and make observational predictions that can be compared to measurements made on a real galaxy sample. You need only choose the pre-built model and simulation snapshot that is appropriate for your science application, and you can immediately generate a Monte Carlo realization of the model with the syntax shown. The syntax in code block 2 shows how to populate the Bolshoi simulation at $z = 0$ with an HOD-style model based on Leauthaud et al. (2011). All Halotools models can populate halo catalogs with mock galaxies using the same syntax shown in code block 2, regardless of the features of the model or the selected halo catalog.

Calling the *populate_mock* method the first time creates the *mock* attribute of a model, and triggers a large amount of pre-processing that need only be done once. The algorithm used to repopulate mock catalogs takes advantage of this pre-processing, so that subsequent calls to *model.mock.populate()* are far faster. For example, the mock population shown in code block 2 based on the Leauthaud et al. (2011) model takes just a few hundred milliseconds on a modern laptop.

The behavior of all Halotools models is controlled by the *param_dict* attribute, a python dictionary where the values of the model parameters are stored. By changing the values of the parameters in the *param_dict*, users can generate alternate mock catalogs based on the updated parameter values. The process of *param_dict* values and repeatedly populating mock catalogs is the typical workflow in an MCMC-type analysis conducted with Halotools.

2.2.3 User-built models

The `empirical_models` sub-package provides far more functionality than a simple set of prebuilt composite models that are ready to generate mock catalogs "out-of-the-box". Halotools has special factory classes that allow users to build their own models connecting galaxies to the dark matter halos that host them. These factories are the foundation of the object-oriented platform that Halotools users can exploit to design their own models of the galaxy-halo connection. This model-building platform is the centerpiece of the `empirical_models` sub-package.

Users choose between a set of component models of the galaxy population and compose them together into a composite model using the appropriate Halotools factory class; HOD-style models are built by the *HodModelFactory* class, subhalo-based models are built by the *SubhaloModelFactory*. Composing together different collections of components gives users a large amount of flexibility to construct highly complex models of galaxy evolution. There are no limits on the number of component models that can be chosen, nor on the number or kind of galaxy population(s) that make up the universe in the

Listing 2: Creating a mock galaxy catalog

```
from halotools.sim_manager import CachedHaloCatalog
halocat = CachedHaloCatalog(simname = 'bolshoi', redshift = 0)

from halotools.empirical_models import PrebuiltHodModelFactory
model = PrebuiltHodModelFactory('leauthaud11')

model.populate_mock(halocat)
print(model.mock.galaxy_table[0:9]) # view the first ten galaxies in the catalog
```

user-defined composite model. Figure 1 shows a cartoon example of how an HOD-style can be built in Halotools.

In choosing component models, users are not restricted to choose from the set of features that ship with the Halotools package. Users are free to write their own component models and use the Halotools factories to build the composite, to write just one new component model and include it in a collection of Halotools-provided components, or anywhere in between. This way, users mostly interested in a specific feature of the galaxy population can focus exclusively on developing code for that one feature, and use existing Halotools components to model the remaining features. The factory design pattern makes it simple to swap out individual features while keeping all other model aspects identical, facilitating users to ask targeted science questions about galaxy evolution and answer these questions via direct computation.

2.3 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 evaluate such 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 extensible 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 without a 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 PACKAGE DEVELOPMENT

3.1 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[3], and the public version of the code is hosted on GitHub[4]. 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

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

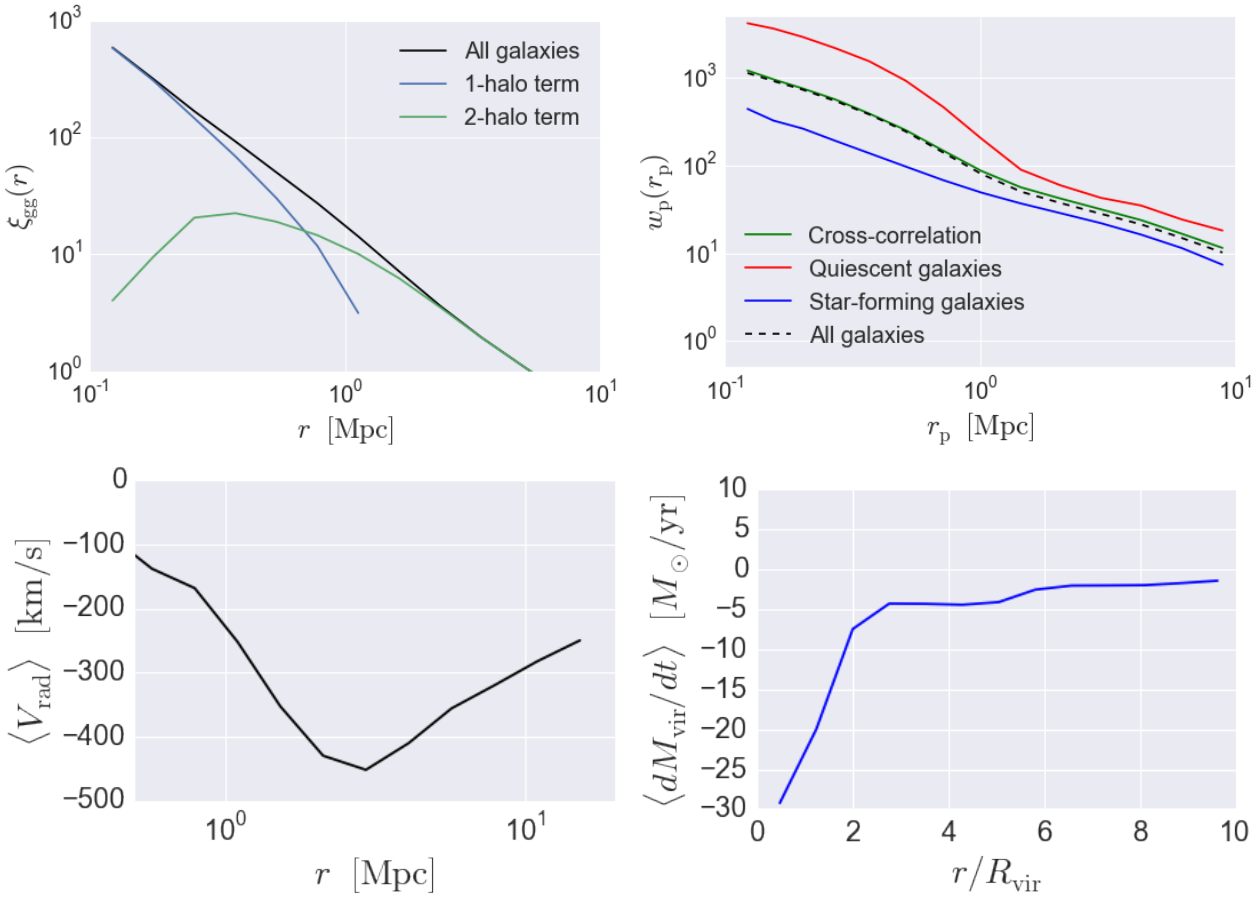

Figure 2. Four example calculations done with Halotools demonstrating the diversity of the `mock_observables` sub-package. Each figure is part of a tutorial found in `http://halotools.readthedocs.io`, to which we refer the reader for details. Here we only point out that each panel demonstrates the result 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.

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`.

3.2 Automated testing

Halotools includes hundreds of unit-tests that are incorporated into the package via the `py.test` framework.[5] 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

[5] `http://pytest.org`

entire test suite passes in all environment configurations. We use `Travis`[6] for continuous integration in Unix environments such as Linux and Mac OS X and `AppVeyor`[7] for Windows environments.

Pull Requests to the `master` branch are additionally subject to a requirement enforced by `Coveralls`.[8] 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.

3.3 Documentation

Documentation of the code base is generated via sphinx[9] and is hosted on ReadTheDocs[10] 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.

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

4 PLANNED FEATURES

5 ACKNOWLEDGMENTS

REFERENCES

Astropy Collaboration Robitaille T. P., Tollerud E. J., Greenfield P., Droettboom M., Bray E., Aldcroft T., et al., 2013, AAP, 558, A33
Behroozi P. S., Conroy C., Wechsler R. H., 2010, ApJ, 717, 379
Behroozi P. S., et al., 2013, ApJ, 763, 18
Behroozi P. S., Wechsler R. H., Wu H.-Y., 2011, ArXiv:1110.4372
Berlind A. A., Weinberg D. H., 2002, ApJ, 575, 587
Conroy C., Wechsler R. H., Kravtsov A. V., 2006, ApJ, 647, 201
Hearin A. P., Watson D. F., 2013, Mon. Not. R. Astron. Soc., 435, 1313
Kravtsov A. V., Berlind A. A., Wechsler R. H., Klypin A. A., Gottlöber S., Allgood B., Primack J. R., 2004, ApJ, 609, 35
Leauthaud A., Tinker J., Behroozi P. S., Busha M. T., Wechsler R. H., 2011, ApJ, 738, 45
Moster B. P., Somerville R. S., Maulbetsch C., van den Bosch F. C., Macciò A. V., Naab T., Oser L., 2010, ApJ, 710, 903
Tinker J. L., Leauthaud A., Bundy K., George M. R., Behroozi P., Massey R., Rhodes J., Wechsler R. H., 2013, ApJ, 778, 93
Yang X., Mo H. J., van den Bosch F. C., 2003, Mon. Not. R. Astron. Soc., 339, 1057