Draft version June 8, 2016
Preprint typeset using LaTeX style AASTeX6 v. 1.0

HIGH-PRECISION FORWARD MODELING OF LARGE-SCALE STRUCTURE: AN OPEN SOURCE APPROACH WITH HALOTOOLS

Andrew P. Hearin[1], Duncan Campbell[2], Erik Tollerud[2,3],
and
Peter Behroozi[15,16], Benedikt Diemer[5], Nathan J. Goldbaum[8], Elise Jennings[13,14], Alexie Leauthaud[9], Yao-Yuan Mao[11, 12], Surhud More[9], John Parejko[4], Manodeep Sinha[6,7], Brigitta Sipocz[4], Andrew Zentner[10]

ABSTRACT

We present the first stable 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 galaxies starting from a catalog of dark matter halos obtained from cosmological simulations. The package supports many of the common forms used to describe such a connection – the halo occupation distribution, the conditional luminosity function, subhalo abundance matching as well as their variants which include effects such as variable galaxy assembly bias. The package includes features which allow satellite galaxies to follow custom number density profiles within their halos, or to have a velocity bias with respect to dark matter, among others. The package has an 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, which can allow a direct comparison to observations. `Halotools` is object-oriented, enabling complex models to be built from a set of simple, interchangeable components, including those of your own creation. `Halotools` has an automated testing suite and is exhaustively documented on http://halotools.readthedocs.io, which includes quickstart guides, source code notes and a large collection of tutorials. The documentation serves as an online textbook on how to build empirical models of galaxy formation with Python.

1. INTRODUCTION

Empirical modeling of the galaxy–halo connection has become a mature subfield of the science of galaxy formation. The modern form of galaxy–halo modeling has been in place for over ten years, including formulations such as the Halo Occupation Distribution (HOD, **?**), the Conditional Luminosity Function (CLF, **?**) and abundance matching (**?**). Because of the computational efficiency and the transparency of the assumptions underlying empirical models, these are the most widely used approaches to incorporating galaxy formation physics into contemporary cosmological likelihood analyses (e.g., **??**). Galaxy–halo models also play a central role in uncovering observational trends that are now considered fundamental to extragalactic astronomy, such as the stellar mass-to-halo mass relation (**?????**) and the demographics of galaxy quenching (**????**).

Building upon now-standard formulations of the galaxy–halo connection, recent advances in this field have seen an increase in the complexity of empirical models. For example, in the abundance matching formulation constrained in **?**, stellar mass is influenced by both present-day halo mass as well as halo concentration. Satellite galaxies in the "delayed-then-rapid" model introduced in **?** have an explicit dependence on the time the satellites first passed within the virial radius of their host halo. In the age matching model (**?**), the star-formation history of both centrals and satellites is tightly coupled to halo mass assembly history across cosmic time. These advances reflect a growing trend in galaxy–halo modeling that we only expect to continue: galaxies *co-evolve* together with their parent halos. While this improves the sophistication of the predictions that can be extracted from empirical models, it also creates additional technical challenges and hurdles for their implementation, especially for beginners.

The standardization of traditional models such as the HOD, CLF and abundance matching provides a natural motivation for a correspondingly standard code base with which the observable predictions of these models can be generated. The trend of galaxy–halo models towards increasing complexity highlights the need for such a code base, particularly one that is open-source and easy to use: it will only become more impractical for new and/or young practitioners to participate at the forefront of this field.

Alongside these scientific developments, over the past several years the Python programming language[1] has become the most popular language in the field of astronomy (**?**). Many aspects of Python make this language well-suited for a fast-moving scientific field: dy-

[1] http://www.python.org

namic typing and automated memory management facilitate rapid development; stable, high-performance libraries such as `Numpy`[2](?) and `Scipy`[3](?) relieve scientists from the need to implement common numerical routines in scientific computing; the Python community has developed a broad toolkit for transparently documenting and testing code in an automated fashion; Python users find its uncluttered syntax to be highly readable and expressive. A significant additional factor leading to the widespread use of Python in astronomy has been the development of `Astropy` (?), a project with the express goal of providing a single core package for astronomy in Python, as well as fostering interoperability between complementary Python astronomy packages.

Motivated by these trends, we present the first stable release of `Halotools` (v0.2), an open-source, object-oriented Python package for building and testing models of the galaxy–halo connection. `Halotools` is community-driven, and already includes contributions from over a dozen scientists spread across numerous universities. Designed with high-speed performance in mind, the package generates mock observations of synthetic galaxy populations with sufficient speed to conduct expansive Monte Carlo Markov Chain (MCMC) likelihood analyses over a diverse and highly customizable set of models.

`Halotools` can be thought of in analogy to Boltzmann-solvers such as `CMBFast` (?), `CAMB` (?) and `CLASS` (?). However, rather than generating ΛCDM predictions for Cosmic Microwave Background power spectra, `Halotools` uses a halo catalog output of a cosmological simulation to predict large-scale structure observables such as galaxy clustering, galaxy–galaxy lensing, void abundance, redshift-space distortions, and related statistics. These predictions are made in the spirit of forward modeling: `Halotools` makes no appeal whatsoever to simulation fitting functions (e.g., ??) that are commonly employed in conventional formulations of the galaxy–halo connection. As these and related fitting functions have well-established upper limits on their accuracy (e.g., ?), our forward modeling approach makes `Halotools` better suited to the needs of the precision cosmology program, **albeit limited to the accuracy and cosmic variance of the underlying numerical simulation**.

`Halotools` is an affiliated package[4] of `Astropy`. Affiliated packages are not part of the `Astropy` core package, but extend the core with typically more domain-specific functionality while maintaining its engineering and interface standards. Since its inception, `Halotools` has been written in fully public view on https://github.com/astropy/halotools, in the spirit of open, reproducible science.

In this paper, we give an overview of the primary features of the `Halotools` package. This paper is not intended to be viewed as code documentation, which is available online at http://halotools.readthedocs.io. Our aim here is to give a simple overview of the features of the package, and to serve as a standard reference for scientists using `Halotools` to support their published work. We outline the basic structure of the code in §??, describe the development workflow in §?? and conclude in §??.

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`.[5]

`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 `sim_manager` sub-package described in §?? 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. The `empirical_models` sub-package described in §?? contains 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 §?? 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.

Although these sub-packages are designed to work 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` provides pre-processed halo catalogs that are science-ready as soon as they are downloaded, use of

[2] http://www.numpy.org

[3] https://www.scipy.org

[4] http://www.astropy.org/affiliated

[5] Cython is a tool that compiles Python-like code into C code. See http://cython.org and ? for further information.

Halotools-provided catalogs is entirely optional and the package works equally well with alternative halo catalogs provided and processed by the user. The empirical_models sub-package can be used to populate mock galaxies into 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 http://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 (??). 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 Figure ??. Cached simulations are stored in the form of an hdf5 file[6](?). 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 located in the cache directory.

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 equivalently 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, multidark and consuelo, for which snapshots at $z = 0, 0.5, 1$ and 2 are available (????). A random downsampling of dark matter particles is also available to accompany each supported snapshot.

2.2. *Empirical Models*

All Halotools models of the galaxy–halo connection are contained in the empirical_models sub-package. Halotools models come in 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. Example component models include the stellar-to-halo mass relation, a Navarro-Frenk-White (NFW) radial profile for satellite distribution or the halo mass-dependence of the quenched fraction. We elaborate upon the structure of Halotools models below, and remind the reader that there are extensive step-by-step guides on this material available at http://halotools.readthedocs.io.

2.2.1. *Model styles*

Halotools composite models come in two different types: HOD-style models and subhalo-based models. In

[6] http://www.h5py.org

Figure 1. Loading a cached halo catalog into memory

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

View the first ten halos in the catalog
>>> print(halocat.halo_table[0:10])

Inspect some of the halo catalog metadata
>>> print(halocat.Lbox, halocat.particle_mass)
```

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 realization of some analytical model. Examples of this approach to the galaxy–halo connection include the HOD (**?**) and CLF (**?**), as well as extensions of these that include additional features such as color-dependence (**?**).

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 (**??**), age matching (**?**), and parameterized stellar-to-halo mass models (**??**).

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. Users need only choose the pre-built model and simulation snapshot that is appropriate for their science application; they can then immediately generate a Monte Carlo realization of the model. The syntax in Figure **??** shows how to populate the Bolshoi simulation at $z = 0$ with an HOD-style model based on **?**. All `Halotools` models can populate halo catalogs with mock galaxies using this same syntax, 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, repopulation of the model shown in Figure **??** based on the **?** 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 varying `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 user-defined composite model. Figure **??** shows a cartoon example of the factory design of an HOD-style model.

In choosing component models, users are not restricted to 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 also 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*

Figure 2. Populating 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)

View the first ten galaxies in the catalog
>>> print(model.mock.galaxy_table[0:10])
```

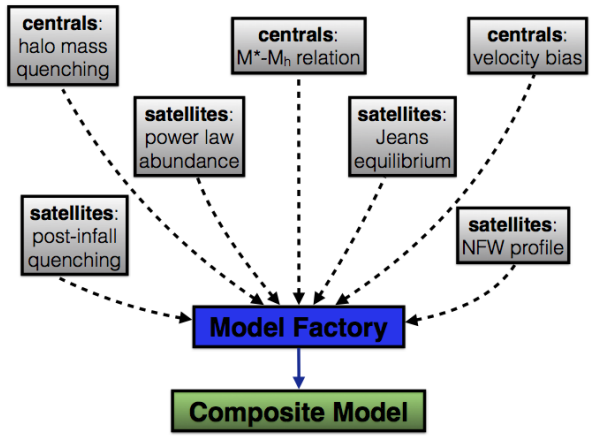

Figure 3. Cartoon example of how an HOD-style model can be built with the `Halotools` factory design pattern; subhalo-based models can be built in a directly analogous fashion. Users select a set of *component model* features of their choosing, and compose them together into a *composite model* using 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, http://halotools.readthedocs.io contains extensive step-by-step tutorials and worked examples demonstrating how to use `Halotools` to model the galaxy–halo connection.

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 spent 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 on http://halotools.readthedocs.io 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.

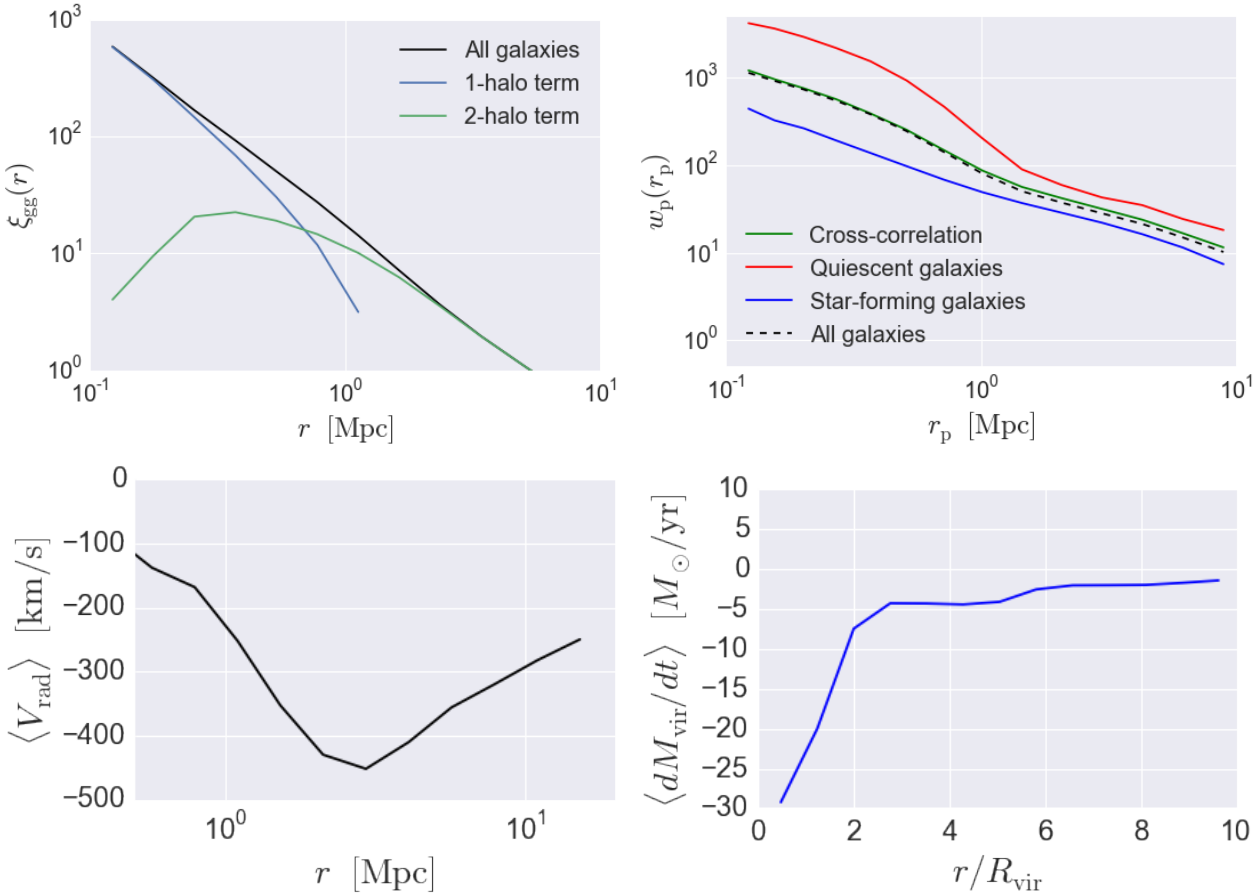

Figure 4. Four example calculations done with `Halotools` demonstrating the diversity of the `mock_observables` sub-package. Each figure is part of a tutorial found in http://halotools.readthedocs.io, to which we refer the reader for details. Here we only point out that each panel demonstrates the result 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.

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`[7], and the public version of the code is hosted on GitHub[8]. 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 and 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 eligible to be merged into `master`.

3.2. *Automated testing*

`Halotools` includes hundreds of unit-tests that are incorporated into the package via the `py.test` framework.[9] These tests are typically small blocks of code that test a specific feature of a specific function. The

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

[8] http://www.github.com

[9] http://pytest.org

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 referring 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. We use `Travis`[10] for continuous integration in Unix environments such as Linux and Mac OS X, and `AppVeyor`[11] for Windows environments.

Pull Requests to the `master` branch are additionally subject to a requirement enforced by `Coveralls`.[12] 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`[13] and is hosted on `ReadTheDocs`[14] at http://halotools.readthedocs.io. The public repository https://github.com/astropy/halotools has a web-hook 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.

4. CONCLUSION

We have presented the first stable release of the `Halotools` package (v0.2), a specialized Python package for building and testing models of the galaxy–halo connection, and analyzing catalogs of dark matter halos. The core functionality of the package includes:

- Fast generation of synthetic galaxy populations using HODs, abundance matching, and related methods.
- Efficient algorithms for calculating galaxy clustering, lensing, z-space distortions, and other astronomical statistics.
- A modular, object-oriented framework for designing galaxy evolution models.
- End-to-end support for reducing halo catalogs and caching them as fast-loading hdf5 files.

We offer several examples below of typical use-cases for which `Halotools` was designed. This list is not intended to be complete, but simply to illustrate the full-featured nature of the package, which includes functionality supporting both galaxy evolution science as well as cosmological simulation analysis. We remind the reader that https://halotools.readthedocs.io contains extensive tutorials and step-by-step guides that can be used as examples for how to carry out each of the example studies below and more.

1. Constrain traditional HOD model parameters with an MCMC-type analysis of observational measurements of projected galaxy clustering $w_{\rm p}(r_{\rm p})$.
2. Build a novel empirical model of galaxy morphology and derive MCMC-type constraints on its parameters using non-traditional large-scale structure measurements such as the morphology-marked correlation function $\mathcal{M}(r_{\rm p})$.

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

[11] https://www.appveyor.com

[12] https://coveralls.io

[13] http://www.sphinx-doc.org

[14] https://readthedocs.io

3. Conduct a study of the connection between the radial profiles of dark matter subhalos and the mass accretion history of their parent halo.

4. Build mock catalogs of galaxies that include forward-models of observational systematics such as intrinsic alignments and/or deblending errors in galaxy shapes.

We would like to conclude this paper with an invitation. As can be seen from the public record of package development on GitHub, contributions of all kinds are warmly welcomed. This could include submitting a Pull Request of a novel feature that has been developed for a scientific publication, or simply submitting a bug report or requesting documentation clarification. It is the hope of the development team that scientists who find the package useful in their own work will also use `Halotools` as an outlet to share their expertise in large-scale structure in a way that can benefit the wider astronomical community.

5. ACKNOWLEDGMENTS

APH would like to express profound gratitude to the Yale Center for Astronomy and Astrophysics, and Meg Urry in particular, for their carte blanche support for this long-term and extremely labor-intensive project. Without the creative freedom offered by the YCAA Prize fellowship, there would be no `Halotools`.

APH gives special thanks to Jim Ford for every last groove of *Harlan County*. APH is grateful to Nikhil Padmanabhan and Frank van den Bosch for invaluable and continuing guidance. DC would like to thank Nikhil Padmanabhan and Frank van den Bosch for their patience and support while developing this package.

We thank the `Astropy` developers for the package-template, which has been critical to the entire process of `Halotools` development. We would like to thank the NumPy, SciPy, IPython, Matplotlib, GitHub, ReadTheDocs, Travis, AppVeyor and Coveralls communities for their extremely useful free software products. We thank all `Halotools` users who have patiently helped improve the package by providing feedback and critiques from the project's alpha-stages to the present. Thanks to Doug Watson and Matt Becker for productive discussions at a formative stage of `Halotools` development, and Yu Feng for help in optimizing the `mock_observables` subpackage and comments on an early draft of this paper.

A portion of this work was also supported by the National Science Foundation under grant PHYS-1066293 and the hospitality of the Aspen Center for Physics. This work was also partially funded by the U.S. National Science Foundation under grant AST 1517563. Support for EJT was provided by NASA through Hubble Fellowship grants #51316.01 awarded by the Space Telescope Science Institute, which is operated by the Association of Universities for Research in Astronomy, Inc., for NASA, under contract NAS 5-26555. PB was supported through program number HST-HF2-51353.001-A, provided by NASA through a Hubble Fellowship grant from STScI, which is operated by the Association of Universities for Research in Astronomy, Incorporated, under NASA contract NAS5-26555. NJG is funded by NSF grant ACI-1535651 as well as by the Gordon and Betty Moore Foundation's Data-Driven Discovery Initiative through Grant GBMF4651. EJ is supported by Fermi Research Alliance, LLC under the U.S. Department of Energy under contract No. DEAC02-07CH11359. SM is supported by Grant-in-Aid for Scientific Research from the JSPS Promotion of Science (15K17600, 16H01089).