abacusutils

Daniel Eisenstein, Philip Pinto, Lehman Garrison, Nina Maksimova

Jun 08, 2022

GETTING STARTED

abacusutils is a package for reading and manipulating data products from the Abacus *N*-body project. In particular, these utilities are intended for use with the AbacusSummit suite of simulations. We provide multiple interfaces: primarily Python 3, but also C/C++ [coming soon!] and language-agnostic interfaces like Unix pipes.

abacusutils is hosted in the abacusorg/abacusutils GitHub repository. Please report bugs and ask questions by opening an issue in that repository.

While you're there, press the GitHub "Watch" button in the top right and select "Custom->Releases" to be notified about bug fixes and new features! This package is still in early stages, and bugs are likely to be identified and squashed, and new performance opportunities identified.

CHAPTER

ONE

INSTALLATION

1.1 Pip Installation

For access to the Python functionality of abacusutils, you can either install via pip or clone from GitHub. The pip installation is recommended if you don't need to modify the source:

```
$ pip install abacusutils
```

This command will also give access to the command-line *Unix Pipes: pipe_asdf* functionality.

Note: Previously, a custom fork of ASDF was required. As of abacusutils 1.0.0, this is no longer required, instead using the extension mechanism of ASDF 2.8.

All the pip-installed functionality is pure-Python, using numba for any performance-intensive routines.

1.2 Installing from Cloned Repository

If you want to hack on the abacusutils source code, then the recommendation is to clone the repo and install the package in pip "editable mode":

```
$ git clone https://github.com/abacusorg/abacusutils.git
$ cd abacusutils
$ pip install -e .[extra]  # install from current dir in editable mode, including extras
```

The `-e` flag ("editable") is optional but recommended so that the installed copy is just a link to the cloned repo (and thus modifications to the Python code will be seen by code that imports abacusutils).

The `.[extra]` syntax says to install from the current directory (`.`), including the set of "optional dependencies" called `extra`. This includes Python packages needed to run things in the `scripts` directory.

Warning: If you first install via pip and then later clone the repo, don't forget to run `pip install -e .[extra]` in the repo. Otherwise, you will have two copies of abacusutils: one cloned, and one installed via pip.

CHAPTER

TWO

USAGE

2.1 Python

abacusutils contains a Python namespace package called `abacusnbody`. This is the name to import (not `abacusutils`, which is just the name of the PyPI package). For example, to import the `compaso_halo_catalog` module, use

```
import abacusnbody.data.compaso_halo_catalog
```

The *Python: compaso_halo_catalog* page has the documentation and API for this module.

The API for the other, less commonly used modules in the `abacusnbody` namespace is located here: *abacusnbody API*.

Specific examples of how to use abacusutils to work with AbacusSummit data will soon be given at the AbacusSummit website: https://abacussummit.readthedocs.io

2.2 C/C++

Coming soon! For simple data access patterns, the *Unix Pipes* approach may suffice.

2.3 Unix Pipes

The `pipe_asdf` Python script reads columns from ASDF files and pipes them to `stdout`. Programs can then read the raw binary from `stdin` without having to worry about the details of file formats or compression. For example, to pipe two columns from `halo_info_000.asdf` to the `client` analysis program, use:

```
$ pipe_asdf halo_info_000.asdf -f N -f x_com | ./client
```

The `pipe_asdf` script is installed when installing abacusutils via pip. Alternatively, it is available directly in the `abacusutils/pipe_asdf/` directory. An example client program is available in the same directory.

See the documentation here: *Unix Pipes: pipe_asdf*.

CHAPTER

THREE

PYTHON: COMPASO_HALO_CATALOG

The `compaso_halo_catalog` module loads halo catalogs from CompaSO, Abacus's on-the-fly halo finder. The module defines one class, `CompaSOHaloCatalog`, whose constructor takes the path to a halo catalog as an argument. Users should use this class as the primary interface to load and manipulate halo catalogs.

The halo catalogs and particle subsamples are stored on disk in ASDF files and are loaded into memory as Astropy tables. Each column of an Astropy table is essentially a Numpy array and can be accessed with familiar Numpy-like syntax. More on Astropy tables here: http://docs.astropy.org/en/stable/table/

Beyond just loading the halo catalog files into memory, this module performs a few other manipulations. Many of the halo catalog columns are stored in bit-packed formats (e.g. floats are scaled to a ratio from 0 to 1 then packed in 16-bit ints), so these columns are unpacked as they are loaded.

Furthermore, the halo catalogs for big simulations are divided across a few dozen files. These files are transparently loaded into one monolithic Astropy table if one passes a directory to `CompaSOHaloCatalog`; to save memory by loading only one file, pass just that file as the argument to `CompaSOHaloCatalog`.

Importantly, because ASDF and Astropy tables are both column- oriented, it can be much faster to load only the subset of halo catalog columns that one needs, rather than all 60-odd columns. Use the `fields` argument to the `CompaSOHaloCatalog` constructor to specify a subset of fields to load. Similarly, the particles can be quite large, and one can use the `subsamples` argument to restrict the particles to the subset one needs.

Some brief examples and technical details about the halo catalog layout are presented below, followed by the full module API. Examples of using this module to work with AbacusSummit data can be found on the AbacusSummit website here: https://abacussummit.readthedocs.io

3.1 Short Example

```
>>> from abacusnbody.data.compaso_halo_catalog import CompaSOHaloCatalog
>>> # Load the RVs and PIDs for particle subsample A
>>> cat = CompaSOHaloCatalog('/storage/AbacusSummit/AbacusSummit_base_c000_ph000/halos/
↪z0.100', subsamples=dict(A=True,pos=True))
>>> print(cat.halos[:5])  # cat.halos is an Astropy Table, print the first 5 rows
   id    npstartA npstartB ... sigmavrad_L2com sigmavtan_L2com rvcirc_max_L2com
-------- -------- -------- ... --------------- --------------- ----------------
25000000        0        2 ...       0.9473971      0.96568024      0.042019103
25000001       11       12 ...      0.86480814       0.8435805      0.046611086
48000000       18       15 ...      0.66734606      0.68342227      0.033434115
58000000       31       18 ...      0.52170926       0.5387341      0.042292822
58001000       38       23 ...       0.4689916      0.40759262      0.034498636
>>> print(cat.halos['N','x_com'][:5])  # print the position and mass of the first 5 halos
  N         x_com [3]
```

(continues on next page)

(continued from previous page)

```
--- ------------------------
278 -998.88525 .. -972.95404
 45  -998.9751 .. -972.88416
101   -999.7485 .. -947.8377
 82    -998.904 .. -937.6313
 43   -999.3252 .. -937.5813
>>> # Examine the particle subsamples associated with the 5th halo
>>> h5 = cat.halos[4]
>>> print(cat.subsamples['pos'][h5['npstartA']:h5['npstartA'] + h5['npoutA']])
        pos [3]
------------------------
  -999.3019 .. -937.5229
 -999.33435 .. -937.5515
-999.38965 .. -937.58777
>>> # At a glance, the pos fields match that of the 5th halo above, so it appears we␣
↪have indexed correctly!
```

3.2 Catalog Structure

The catalogs are stored in a directory structure that looks like:

```
- SimulationName/
    - halos/
        - z0.100/
            - halo_info/
                halo_info_000.asdf
                halo_info_001.asdf
                ...
            - halo_rv_A/
                halo_rv_A_000.asdf
                halo_rv_A_001.asdf
                ...
            - <field & halo, rv & PID, subsample A & B directories>
        - <other redshift directories, some with particle subsamples, others without>
```

The file numbering roughly corresponds to a planar chunk of the simulation (all y and z for some range of x). The matching of the halo_info file numbering to the particle file numbering is important; halo_info files index into the corresponding particle files.

The halo catalogs are stored on disk in ASDF files (https://asdf.readthedocs.io/). The ASDF files start with a human-readable header that describes the data present in the file and metadata about the simulation from which it came (redshift, cosmology, etc). The rest of the file is binary blobs, each representing a column (i.e. a halo property).

Internally, the ASDF binary portions are usually compressed. This should be transparent to users, although you may be prompted to install the blosc package if it is not present. Decompression should be fast, at least 500 MB/s per core.

3.3 Particle Subsamples

We define two disjoint sets of "subsample" particles, called "subsample A" and "subsample B". Subsample A is a few percent of all particles, with subsample B a few times larger than that. Particles membership in each group is a function of PID and is thus consistent across redshift.

At most redshifts, we only output halo catalogs and halo subsample particle PIDs. This aids with construction of merger trees. At a few redshifts, we provide subsample particle positions as well as PIDs, for both halo particles and non-halo particles, called "field" particles. Only halo particles (specifically, L1 particles) may be loaded through this module; field particles and L0 halo particles can be loaded by reading the particle files directly with *read_abacus module*.

Use the `subsamples` argument to the constructor to specify loading subsample A and/or B, and which fields—pos, vel, pid—to load. Note that if only one of pos & vel is specified, the IO amount is the same, because the pos & vel are packed together in *RVint format*. But the memory usage and time to unpack will be lower.

3.4 Halo File Types

Each file type (for halos, particles, etc) is grouped into a subdirectory. These subdirectories are:

- `halo_info/` The primary halo catalog files. Contains stats like CoM positions and velocities and moments of the particles. Also indicates the index and count of subsampled particles in the `halo_pid_A/B` and `halo_rv_A/B` files.
- `halo_pid_A/` and `halo_pid_B/` The 64-bit particle IDs of particle subsamples A and B. The PIDs contain information about the Lagrangian position of the particles, whether they are tagged, and their local density.

The following subdirectories are only present for the redshifts for which we output particle subsamples and not just halo catalogs:

- `halo_rv_A/` and `halo_rv_B/` The positions and velocities of the halo subsample particles, in "RVint" format. The halo associations are recoverable with the indices in the `halo_info` files.
- `field_rv_A/` and `field_rv_B/` Same as `halo_rv_<A|B>/`, but only for the field (non-halo) particles.
- `field_pid_A/` and `field_pid_B/` Same as `halo_pid_<A|B>/`, but only for the field (non-halo) particles.

3.5 Bit-packed Formats

The "RVint" format packs six fields (x,y,z, and vx,vy,vz) into three ints (12 bytes). Positions are stored to 20 bits (global), and velocities 12 bits (max 6000 km/s).

The PIDs are 8 bytes and encode a local density estimate, tag bits for merger trees, and a unique particle id, the last of which encodes the Lagrangian particle coordinate.

These are described in more detail on the AbacusSummit Data Model page.

Use the `unpack_bits` argument of the `CompaSOHaloCatalog` constructor to specify which PID bit fields you want unpacked. Be aware that some of them might use a lot of memory; e.g. the Lagrangian positions are three 4-byte floats per subsample particle. Also be aware that some of the returned arrays use narrow int dtypes to save memory, such as the `lagr_idx` field using `int16`. It is easy to silently overflow such narrow int types; make sure your operations stay within the type width and cast if necessary.

3.6 Field Subset Loading

Because the ASDF files are column-oriented, it is possible to load just one or a few columns (halo catalog fields) rather than the whole file. This can save huge amounts of IO, memory, and CPU time (the latter due to the decompression). Use the `fields` argument to the `CompaSOHaloCatalog` constructor to specify the list of columns you want.

In detail, some columns are stored as ratios to other columns. For example, `r90` is stored as a ratio relative to `r100`. So to properly unpack `r90`, the `r100` column must also be read. `CompaSOHaloCatalog` knows about these dependencies and will load the minimum set necessary to return the requested columns to the user. However, this may result in more IO than expected. The `verbose` constructor flag or the `dependency_info` field of the `CompaSOHaloCatalog` object may be useful for diagnosing exactly what data is being loaded.

Despite the potential extra IO and CPU time, the extra memory usage is granular at the level of individual files. In other words, when loading multiple files, the concatenated array will never be constructed for columns that only exist for dependency purposes.

3.7 Superslab (Chunk) Processing

The halo catalogs are divided across multiple files, called "superslabs", which are typically planar chunks of the simulation volume (all y,z for some range of x, with a bit of overlap at the boundaries). Applications that can process the volume superslab-by-superslab can save a substantial amount of memory compared to loading the full volume. To load a single superslab, pass the corresponding `halo_info_XXX.asdf` file as the `path` argument:

```
cat = CompaSOHaloCatalog('AbacusSummit_base_c000_ph000/halos/z0.100/halo_info/halo_info_
↪000.asdf')
```

If your application needs one slab of padding, you can pass a list of files and proceed in a rolling fashion:

```
cat = CompaSOHaloCatalog(['AbacusSummit_base_c000_ph000/halos/z0.100/halo_info/halo_info_
↪033.asdf',
                          'AbacusSummit_base_c000_ph000/halos/z0.100/halo_info/halo_info_
↪000.asdf',
                          'AbacusSummit_base_c000_ph000/halos/z0.100/halo_info/halo_info_
↪001.asdf'])
```

3.8 Superslab Filtering

Another way to save memory is to use the `filter_func` argument. This function will be called for each superslab, and must return a mask representing the rows to keep. For example, to drop all halos with less than 100 particles, use:

```
cat = CompaSOHaloCatalog(..., filter_func=lambda h: h['N'] >= 100)
```

Because this mask is applied on each superslab immediately after loading, the full, unfiltered table is never constructed, thus saving memory.

The filtering adds some CPU time, but in many cases loading catalogs is IO limited, so this won't add much overhead.

3.9 Multi-threaded Decompression

The Blosc compression we use inside the ASDF files supports multi-threaded decompression. We have packed AbacusSummit files with 4 Blosc blocks (each ~few MB) per ASDF block, so 4 Blosc threads is probably the optimal value. This is the default value, unless fewer cores are available (as determined by the process affinity mask).

Note: Loading a CompaSOHaloCatalog will use 4 decompression threads by default.

You can control the number of decompression threads with:

```
import abacusnbody.data.asdf
abacusnbody.data.asdf.set_nthreads(N)
```

3.10 API

class abacusnbody.data.compaso_halo_catalog.**CompaSOHaloCatalog**(*path*, *cleaned=True*, *subsamples=False*, *convert_units=True*, *unpack_bits=False*, *fields='DEFAULT_FIELDS'*, *verbose=False*, *cleandir=None*, *filter_func=None*, *halo_lc=None*, ***kwargs*)

Bases: `object`

A halo catalog from Abacus's on-the-fly group finder.

__init__(*path*, *cleaned=True*, *subsamples=False*, *convert_units=True*, *unpack_bits=False*, *fields='DEFAULT_FIELDS'*, *verbose=False*, *cleandir=None*, *filter_func=None*, *halo_lc=None*, ***kwargs*)

Loads halos. The `halos` field of this object will contain the halo records; and the `subsamples` field will contain the corresponding halo/field subsample positions and velocities and their ids (if requested via `subsamples`). The `header` field contains metadata about the simulation.

Whether a particle is tagged or not is returned when loading the halo and field pids, as it is encoded for each in the 64-bit PID. The local density of the particle is also encoded in the PIDs and returned upon loading those.

Parameters

- **path** (*path-like or list of path-like*) – The halo catalog directory, like `MySimulation/halos/z1.000/`. Or a single halo info file, or a list of halo info files. Will accept `halo_info` dirs or "redshift" dirs (e.g. `z1.000/halo_info/` or `z1.000/`).

  Note: To load cleaned catalogs, you do *not* need to pass a different argument to the `path` directory. Use `cleaned=True` instead and the path to the cleaning info will be detected automatically (or see `cleandir`).

- **cleaned** (*bool, optional*) – Loads the "cleaned" version of the halo catalogues. Always recommended. Assumes there is a directory called `cleaning/` at the same level as

the top-level simulation directory (or see `cleandir`). Default: True. False returns the out-of-the-box CompaSO halos. May be useful for specific applications.

- **subsamples** (*bool or dict, optional*) – Load halo particle subsamples. True or False may be specified to load all particles or none, or a dict to specify whether to load subsample A and/or B, with pos, vel, and/or pid fields:

```
subsamples=dict(A=True, B=True, pos=True, vel=True, pid=True)
```

  The `rv` key may be used as shorthand to set both `pos` and `vel`. False (the default) loads nothing.

- **convert_units** (*bool, optional*) – Convert positions from unit-box units to BoxSize-box units, velocities already come in km/s. Default: True.

- **unpack_bits** (*bool, or list of str, optional*) – Extract information from the PID field of each subsample particle info about its Lagrangian position, whether it is tagged, and its current local density. If False, only the particle ID part will be extracted. Note that this per-particle information can be large. Can be a list of str, in which case only those fields will be unpacked. Field names are: ('pid', 'lagr_pos', 'tagged', 'density', 'lagr_idx'). Default: False.

- **fields** (*str or list of str, optional*) – A list of field names/halo properties to load. Selecting a small subset of fields can be substantially faster than loading all fields because the file IO will be limited to the desired fields. See `compaso_halo_catalog.user_dt` or the AbacusSummit Data Model page for a list of available fields. See `compaso_halo_catalog.clean_dt` for the list of cleaned halo fields that will be loaded. 'all' will also load main progenitor information, which could be slow. Default: 'DEFAULT_FIELDS'

- **verbose** (*bool, optional*) – Print informational messages. Default: False

- **cleandir** (*str, optional*) – Where the halo catalog cleaning files are located (usually called `cleaning/`). Default of None will try to detect it automatically. Only has any effect if using `cleaned=True`.

- **filter_func** (*function, optional*) – A mask function to be applied to each superslab as it is loaded. The function must take one argument (a halo table) and return a boolean array or similar mask on the rows. Simple lambda expressions are particularly useful here; for example, to load all halos with 100 particles or more, use:

```
filter_func=lambda h: h['N'] >= 100
```

- **halo_lc** (*bool or None, optional*) – Whether the catalog is a halo light cone catalog, i.e. an output of the CompaSO halo light cone pipeline. Default of None means to detect based on the catalog path.

nbytes(*halos=True*, *subsamples=True*)

Return the memory usage of the big arrays: the halo catalog and the particle subsamples

static is_path_halo_lc(*path*)

abacusnbody.data.compaso_halo_catalog.**join_arrays**(*offset*, *array_original*, *array_merge*, *array_joined*, *npstart_original*, *npout_original*, *npstart_merge*, *npout_merge*, *np_total*)

abacusnbody.data.compaso_halo_catalog.**unpack_euler16**(*bin_this*)

CHAPTER

FOUR

ABACUSHOD

The AbacusHOD module loads halo catalogs from the AbacusSummit simulations and outputs multi-tracer mock galaxy catalogs. The code is highly efficient and contains a large set of HOD extensions such as secondary biases (assembly biases), velocity biases, and satellite profile flexibilities. The baseline HODs are based on those from Zheng et al. 2007 and Alam et al. 2020. The HOD extensions are first explained in Yuan et al. 2018, and more recently summarized in Yuan et al. 2020b . This HOD code also supports RSD and incompleteness. The code is fast, completeling a $(2Gpc/h)^3$ volume in 80ms per tracer on a 32 core desktop system, and the performance should be scalable. The module also provides efficient correlation function and power spectrum calculators. This module is particularly suited for efficiently sampling HOD parameter space. We provide examples of docking it onto `emcee` and `dynesty` samplers.

The module defines one class, `AbacusHOD`, whose constructor takes the path to the simulation volume, and a set of HOD parameters, and runs the `staging` function to compile the simulation halo catalog as a set of arrays that are saved on memory. The `run_hod` function can then be called to generate galaxy catalogs.

The output takes the format of a dictionary of dictionaries, where each subdictionary corresponds to a different tracer. Currently, we have enabled tracer types: LRG, ELG, and QSO. Each subdictionary contains all the mock galaxies of that tracer type, recording their properties with keys `x`, `y` , `z`, `vx`, `vy`, `vz`, `mass`, `id`, `Ncent`. The coordinates are in Mpc/h, and the velocities are in km/s. The `mass` refers to host halo mass and is in units of Msun/h. The `id` refers to halo id, and the `Ncent` key refers to number of central galaxies for that tracer. The first `Ncent` galaxies in the catalog are always centrals and the rest are satellites.

The galaxies can be written to disk by setting the `write_to_disk` flag to `True` in the argument of `run_hod`. However, the I/O is slow and the``write_to_disk`` flag defaults to `False`.

The core of the AbacusHOD code is a two-pass memory-in-place algorithm. The first pass of the halo+particle subsample computes the number of galaxies generated in total. Then an empty array for these galaxies is allocated in memory, which is then filled on the second pass of the halos+particles. Each pass is accelerated with numba parallel. The default threading is set to 16.

4.1 Theory

The baseline HOD for LRGs comes from Zheng et al. 2007:

$$\bar{n}_{\rm cent}(M) = \frac{1}{2}{\rm erfc}\left[\frac{\ln(M_{\rm cut}/M)}{\sqrt{2}\sigma}\right],$$

$$\bar{n}_{\rm sat}(M) = \left[\frac{M-\kappa M_{\rm cut}}{M_1}\right]^{\alpha}\bar{n}_{\rm cent}(M),$$

The baseline HOD for ELGs and QSOs comes from Alam et al. 2020. The actual calculation is complex and we refer the readers to section 3 of said paper for details.

In the baseline implementation, the central galaxy is assigned to the center of mass of the halo, with the velocity vector also set to that of the center of mass of the halo. Satellite galaxies are assigned to particles of the halo with equal

weights. When multiple tracers are enabled, each halo/particle can only host a single tracer type. However, we have not yet implemented any prescription of conformity.

The secondary bias (assembly bias) extensions follow the recipes described in Xu et al. 2020 , where the secondary halo property (concentration or local overdensity) is directly tied to the mass parameters in the baseline HOD ($M_{\rm cut}$ and M_1):

$$\log_{10} M_{\rm cut}^{\rm mod} = \log_{10} M_{\rm cut} + A_c(c^{\rm rank} - 0.5) + B_c(\delta^{\rm rank} - 0.5)$$

$$\log_{10} M_{1}^{\rm mod} = \log_{10} M_{1} + A_s(c^{\rm rank} - 0.5) + B_s(\delta^{\rm rank} - 0.5)$$

where c and δ represent the halo concentration and local overdensity, respectively. These secondary properties are ranked within narrow halo mass bins, and the rank are normalized to range from 0 to 1, as noted by the $\rm rank$ superscript. (A_c, B_c, A_s, B_s) form the four parameters describing secondary biases in the HOD model. The default for these parameters are 0.

The velocity bias extension follows the common prescription as described in Guo et al. 2015 .

$$\sigma_c = \alpha_c \sigma_h$$

$$v_s - v_h = \alpha_s (v_p - v_h)$$

where the central velocity bias parameter α_c sets the ratio of central velocity dispersion vs. halo velocity dispersion. The satellite velocity bias parameter α_c sets the ratio between the satellite peculiar velocity to the particle peculiar velocity. The default for these two parameters are 0 and 1, respectively.

We additionaly introduce a set of satellite profile parameters (s, s_v, s_p, s_r) that allow for flexibilities in how satellite galaxies are distributed within a halo. They respecctively allow the galaxy weight per particle to depend on radial position (s), peculair velocity (s_v), perihelion distance of the particle orbit (s_p), and the radial velocity (s_v). The default values for these parameters are 0. A detailed description of these parameters are available in Yuan et al. 2018, and more recently in Yuan et al. 2020b .

Some brief examples and technical details about the module layout are presented below, followed by the full module API.

4.2 Short Example

The first step is to create the configuration file such as `config/abacus_hod.yaml`, which provides the full customizability of the HOD code. By default, it lives in your current work directory under a subdirectory `./config`. A template with default settings are provided under `abacusutils/scripts/config`.

With the first use, you should define which simulation box, which redshift, the path to simulation data, the path to output datasets, the various HOD flags and an initial set of HOD parameters. Other decisions that need to be made initially (you can always re-do this but it would take some time) include: do you only want LRGs or do you want other tracers as well? Do you want to enable satellite profile flexibilities (the s, s_v, s_p, s_r parameters)? If so, you need to turn on `want_ranks` flag in the config file. If you want to enable secondary bias, you need to set `want_AB` flag to true in the config file. The local environment is defined by total mass within 5 Mpc/h but beyond `r98`.

IMPORTANT: Running this code is a two-part process. First, you need to run the `prepare_sim` code, which generates the necessary data files for that simulation. Then you can run the actual HOD code. The first step only needs to be done once for a simulation box, but it can be slow, depending on the downsampling and the features you choose to enable.

So first, you need to run the `prepare_sim` script, this extracts the simulation outputs and organizes them into formats that are suited for the HOD code. This code can take approximately an hour depending on your configuration settings and system capabilities. We recommend setting the `Nthread_load` parameter to `min(sys_core_count, memoryGB_divided_by_30)`. You can run `load_sims` on command line with

```
python -m abacusnbody.hod.prepare_sim --path2config PATH2CONFIG
```

Within Python, you can run the same script with `from abacusnbody.hod import prepare_sim` and then `prepare_sim.main(/path/to/config.yaml)`.

If your config file lives in the default location, i.e. `./config`, then you can ignore the `-path2config` flag. Once that is finished, you can construct the `AbacusHOD` object and run fast HOD chains. A code template is given in `abacusutils/scripts/run_hod.py` for running a few example HODs and `abacusutils/scripts/run_emcee.py` for integrating with the `emcee` sampler.

To use the given `run_hod.py` script to run a custom configuration file, you can simply run the given script in bash

```
python run_hod.py --path2config PATH2CONFIG
```

You can also consruct the AbacusHOD object yourself within Python and run HODs from there. Here we show the scripts within `run_hod.py` for reference.:

```
import os
import glob
import time
import yaml
import numpy as np
import argparse

from abacusnbody.hod.abacus_hod import AbacusHOD

path2config = 'config/abacus_hod.yaml' # path to config file

# load the config file and parse in relevant parameters
config = yaml.safe_load(open(path2config))
sim_params = config['sim_params']
HOD_params = config['HOD_params']
clustering_params = config['clustering_params']

# additional parameter choices
want_rsd = HOD_params['want_rsd']
write_to_disk = HOD_params['write_to_disk']

# create a new AbacusHOD object
newBall = AbacusHOD(sim_params, HOD_params, clustering_params)

# first hod run, slow due to compiling jit, write to disk
mock_dict = newBall.run_hod(newBall.tracers, want_rsd, write_to_disk, Nthread = 16)

# run the 10 different HODs for timing
for i in range(10):
    newBall.tracers['LRG']['alpha'] += 0.01
    print("alpha = ",newBall.tracers['LRG']['alpha'])
    start = time.time()
    mock_dict = newBall.run_hod(newBall.tracers, want_rsd, write_to_disk = False,␣
↪Nthread = 64)
    print("Done iteration ", i, "took time ", time.time() - start)
```

The class also provides fast 2PCF calculators. For example to compute the redshift-space 2PCF ($\xi(r_p, \pi)$):

```
# load the rp pi binning from the config file
bin_params = clustering_params['bin_params']
rpbins = np.logspace(bin_params['logmin'], bin_params['logmax'], bin_params['nbins'])
pimax = clustering_params['pimax']
pi_bin_size = clustering_params['pi_bin_size']    # the pi binning is configrured by pi_
↪max and bin size

mock_dict = newBall.run_hod(newBall.tracers, want_rsd, write_to_disk)
xirppi = newBall.compute_xirppi(mock_dict, rpbins, pimax, pi_bin_size)
```

4.3 Light Cones

AbacusHOD supports generating HOD mock catalogs from halo light cone catalogs (PR #28). Details on the usage will be provided here soon.

Notes

Currently, when RSD effects are enabled in the HOD code for the halo light cones, the factor `velz2kms`, determining the size of the RSD correction to the position along the line of sight, is the same for all galaxies at a given redshift catalog.

4.4 API

abacusnbody.hod.abacus_hod.**searchsorted_parallel**(*a*, *b*)

class abacusnbody.hod.abacus_hod.**AbacusHOD**(*sim_params*, *HOD_params*, *clustering_params=None*, *chunk=- 1*, *n_chunks=1*)

Bases: `object`

A highly efficient multi-tracer HOD code for the AbacusSummmit simulations.

__init__(*sim_params*, *HOD_params*, *clustering_params=None*, *chunk=- 1*, *n_chunks=1*)

Loads simulation. The `sim_params` dictionary specifies which simulation volume to load. The `HOD_params` specifies the HOD parameters and tracer configurations. The `clustering_params` specifies the summary statistics configurations. The `HOD_params` and `clustering_params` can be set to their default values in the `config/abacus_hod.yaml` file and changed later. The `sim_params` cannot be changed once the `AbacusHOD` object is created.

Parameters

- **sim_params** (*dict*) –

  Dictionary of simulation parameters. Load from `config/abacus_hod.yaml`. The dictionary should contain the following keys:

  - `sim_name`: str, name of the simulation volume, e.g. 'AbacusSummit_base_c000_ph006'.
  - `sim_dir`: str, the directory that the simulation lives in, e.g. '/path/to/AbacusSummit/'.
  - `output_dir`: str, the diretory to save galaxy to, e.g. '/my/output/galalxies'.

  - subsample_dir: str, where to save halo+particle subsample, e.g. '/my/output/subsamples/'.
  - z_mock: float, which redshift slice, e.g. 0.5.

- **HOD_params** (*dict*) –

  HOD parameters and tracer configurations. Load from config/abacus_hod.yaml. It contains the following keys:

  - **tracer_flags: dict, which tracers is enabled:**
    - LRG: bool, default True.
    - ELG: bool, default False.
    - QSO: bool, default False.
  - want_ranks: bool, enable satellite profile flexibilities. If False, satellite profile follows the DM, default True.
  - want_rsd: bool, enable RSD? default True. # want RSD?
  - Ndim: int, grid density for computing local environment, default 1024.
  - density_sigma: float, scale radius in Mpc / h for local density definition, default 3.
  - write_to_disk: bool, output to disk? default False. Setting to True decreases performance.
  - LRG_params: dict, HOD parameter values for LRGs. Default values are given in config file.
  - ELG_params: dict, HOD parameter values for ELGs. Default values are given in config file.
  - QSO_params: dict, HOD parameter values for QSOs. Default values are given in config file.

- **clustering_params** (*dict, optional*) –

  Summary statistics configuration parameters. Load from config/abacus_hod.yaml. It contains the following keys:

  - clustering_type: str, which summary statistic to compute. Options: wp, xirppi, default: xirppi.
  - **bin_params: dict, transverse scale binning.**
    - logmin: float, $\log_{10} r_{\min}$ in Mpc/h.
    - logmax: float, $\log_{10} r_{\max}$ in Mpc/h.
    - nbins: int, number of bins.
  - pimax: int, $\pi_{\max}$.
  - pi_bin_size: int, size of bins along of the line of sight. Need to be divisor of pimax.

- **chunk** (*int, optional*) – Index of current chunk. Must be between 0 and n_chunks-1. Files associated with this chunk are written out as {tracer}s_{chunk}.dat. Default is -1 (no chunking).

- **n_chunks** (*int, optional*) – Number of chunks to split the input from the halo+particle subsample and number of output files in which to write out the galaxy catalogs following the format {tracer}s_{chunk}.dat.

staging()

Constructor call this function to load the halo+particle subsamples onto memory.

run_hod(*tracers=None*, *want_rsd=True*, *reseed=None*, *write_to_disk=False*, *Nthread=16*, *verbose=False*, *fn_ext=None*)

Runs a custom HOD.

Parameters

- **tracers** (*dict*) – dictionary of multi-tracer HOD. `tracers['LRG']` is the dictionary of LRG HOD parameters, overwrites the `LRG_params` argument in the constructor. Same for keys `'ELG'` and `'QSO'`.
- **want_rsd** (*bool*) – enable RSD? default `True`.
- **reseed** (*int*) – re-generate random numbers? supply random number seed. This overwrites the pre-generated random numbers, at a performance cost. Default `None`.
- **write_to_disk** (*bool*) – output to disk? default `False`. Setting to `True` decreases performance.
- **Nthread** (*int*) – number of threads in the HOD run. Default 16.
- **verbose** (*bool,*) – detailed stdout? default `False`.
- **fn_ext** (*str*) – filename extension for saved files. Only relevant when `write_to_disk = True`.

Returns

mock_dict – dictionary of galaxy outputs. Contains keys `'LRG'`, `'ELG'`, and `'QSO'`. Each tracer key corresponds to a sub-dictionary that contains the galaxy properties with keys `'x'`, `'y'`, `'z'`, `'vx'`, `'vy'`, `'vz'`, `'mass'`, `'id'`, `Ncent'`. The coordinates are in Mpc/h, and the velocities are in km/s. The `'mass'` refers to host halo mass and is in units of Msun/h. The `'id'` refers to halo id, and the `'Ncent'` key refers to number of central galaxies for that tracer. The first `'Ncent'` galaxies in the catalog are always centrals and the rest are satellites.

Return type

dict

compute_ngal(*tracers=None*, *Nthread=16*)

Computes the number of each tracer generated by the HOD

Parameters

- **tracers** (*dict*) – dictionary of multi-tracer HOD. `tracers['LRG']` is the dictionary of LRG HOD parameters, overwrites the `LRG_params` argument in the constructor. Same for keys `'ELG'` and `'QSO'`.
- **Nthread** (*int*) – Number of threads in the HOD run. Default 16.

Returns

- **ngal_dict** (*dict*)
- *dictionary of number of each tracer.*
- **fsat_dict** (*dict*)
- *dictionary of satellite fraction of each tracer.*

compute_clustering(*mock_dict*, **args*, ***kwargs*)

Computes summary statistics, currently enabling `wp` and `xirppi`.

Parameters

- **mock_dict** (*dict*) – dictionary of tracer positions. Output of run_hod.
- **Ntread** (*int*) – number of threads in the HOD run. Default 16.
- **rpbins** (*np.array*) – array of transverse bins in Mpc/h.
- **pimax** (*int*) – maximum bin edge along the line of sight direction, in Mpc/h.
- **pi_bin_size** (*int*) – size of bin along the line of sight. Currently, we only support linear binning along the line of sight.

Returns

clustering – dicionary of summary statistics. Auto-correlations/spectra can be accessed with keys such as 'LRG_LRG'. Cross-correlations/spectra can be accessed with keys such as 'LRG_ELG'.

Return type

dict

compute_xirppi(*mock_dict*, *rpbins*, *pimax*, *pi_bin_size*, *Nthread=8*)

Computes $\xi(r_p, \pi)$.

Parameters

- **mock_dict** (*dict*) – dictionary of tracer positions. Output of run_hod.
- **Ntread** (*int*) – number of threads in the HOD run. Default 16.
- **rpbins** (*np.array*) – array of transverse bins in Mpc/h.
- **pimax** (*int*) – maximum bin edge along the line of sight direction, in Mpc/h.
- **pi_bin_size** (*int*) – size of bin along the line of sight. Currently, we only support linear binning along the line of sight.

Returns

clustering – dicionary of summary statistics. Auto-correlations/spectra can be accessed with keys such as 'LRG_LRG'. Cross-correlations/spectra can be accessed with keys such as 'LRG_ELG'.

Return type

dict

compute_multipole(*mock_dict*, *rpbins*, *pimax*, *Nthread=8*)

compute_wp(*mock_dict*, *rpbins*, *pimax*, *pi_bin_size*, *Nthread=8*)

Computes w_p.

Parameters

- **mock_dict** (*dict*) – dictionary of tracer positions. Output of run_hod.
- **Ntread** (*int*) – number of threads in the HOD run. Default 16.
- **rpbins** (*np.array*) – array of transverse bins in Mpc/h.
- **pimax** (*int*) – maximum bin edge along the line of sight direction, in Mpc/h.
- **pi_bin_size** (*int*) – size of bin along the line of sight. Currently, we only support linear binning along the line of sight.

Returns

clustering – dicionary of summary statistics. Auto-correlations/spectra can be accessed with keys such as 'LRG_LRG'. Cross-correlations/spectra can be accessed with keys such as 'LRG_ELG'.

Return type
dict

gal_reader(*output_dir=None*, *simname=None*, *sim_dir=None*, *z_mock=None*, *want_rsd=None*, *tracers=None*)

Loads galaxy data given directory and return a mock_dict dictionary.

Parameters

- **sim_name** (*str*) – name of the simulation volume, e.g. 'AbacusSummit_base_c000_ph006'.
- **sim_dir** (*str*) – the directory that the simulation lives in, e.g. '/path/to/AbacusSummit/'.
- **output_dir** (*str*) – the diretory to save galaxy to, e.g. '/my/output/galalxies'.
- **z_mock** (*floa*) – which redshift slice, e.g. 0.5.
- **want_rsd** (*bool*) – RSD?
- **tracers** (*dict*) – dictionary of tracer types to load, e.g. *{'LRG', 'ELG'}*.

Returns
``**mock_dict**`` – dictionary of tracer positions. Output of run_hod.

Return type
dict

CHAPTER

FIVE

UNIX PIPES: PIPE_ASDF

`pipe_asdf` is a Python script to unpack Abacus ASDF files (such as halo catalog or particle data) and write them out via a Unix pipe (stdout). The intention is to provide a simple way for C, C++, Fortran, etc, codes to read ASDF files while letting Python handle the details of the file formats, compression, and other things Python does well.

5.1 Usage

```
pipe_asdf [-h] [-f FIELD] [--nthread NTHREAD] asdf-file [asdf-file ...] | ./client
```

5.1.1 positional arguments

asdf-file
An ASDF file. Multiple may be specified.

5.1.2 optional arguments

-h, --help show this help message and exit

-f FIELD, --field FIELD A field/column to pipe. Multiple -f flags are allowed, in which case fields will be piped in the order they are specified. (default: None)

--nthread NTHREAD Number of blosc decompression threads (when applicable). For AbacusSummit, use 1 to 4. (default: 4)

5.2 Binary Format of Piped Data

The binary format of the piped data is simple:

1) an 8-byte int indicating the number of data values
2) a 4-byte int indicating the width of the primitive data type that composes the data (e.g. 4 for float, 8 for double). Largely provided as a sanity check.
3) the data, consisting of a number of bytes equal to the product of the preceeding ints
4) Repeat from (1) for all fields requested

So the expected pattern for the client code is to read the int64 and int32, take the product, allocate that many bytes, then read the data into that allocation.

When passing multiple files, a single column will be read from all files before moving to the next column. In other words, the client sees the concatenated data.

From a performance perspective, the pipe operation probably amounts to a memcpy. So a small performance hit, but likely vanishingly small compared to the actual IO and analysis.

Ultimately, this pipe scheme is not a replacement for direct access to the files, but it may be helpful for applications with simple data access patterns.

5.3 Entry Points

Technically, `pipe_asdf` is a "console script" alias provided by setuptools to invoke the `abacusnbody.data.pipe_asdf` module as a script. This alias is usually installed in a user's PATH environment variable when installing abacusutils via pip, but if not, one could equivalently invoke the script with:

```
$ python3 -m abacusnbody.data.pipe_asdf
```

The `abacusnbody/pipe_asdf` directory also contains a symlink to this file, so from this directory one can also run

```
$ ./pipe_asdf.py
```

5.4 To-do

- Add a "-k/–key" flag to read header fields. Decide on a wire protocol.
- Add CompaSOHaloCatalog hooks to pipe the unpacked data (?)

5.5 Python API

abacusnbody.data.pipe_asdf.**unpack_to_pipe**(*asdf_fns*, *fields*, *data_key='data'*, *header_key='header'*, *pipe=<_io.BufferedWriter name='<stdout>'>*, *nthread=4*, *verbose=True*)

abacusnbody.data.pipe_asdf.**main**()

Invoke the command-line interface

5.6 Example C Client Code

An example C program called `client.c` that receives data over a pipe is given in the abacusutils/pipe_asdf directory.

From this directory, one can build the `client` program by running

```
$ make
```

and run it with:

```
$ ./pipe_asdf.py halo_info_000.asdf -f N -f x_com | ./client
```

You can use the example `halo_info_000.asdf` file symlinked in the `pipe_asdf` directory to test this.

This program is a stand-in for an analysis code. In this case, it just reads the raw binary data for two columns, `N` and `x_com`, and prints the values.

CHAPTER

SIX

SIMULATION METADATA

The `abacusnbody.metadata` module contains the parameters and states for Abacus simulations like AbacusSummit. One can use this module to query information about simulations without actually downloading or opening any simulation files.

The main entry point is the `get_meta(simname, redshift=z)` function. Examples and the API are below.

6.1 Examples

6.1.1 Omega(z)

```
import abacusnbody.metadata
meta = abacusnbody.metadata.get_meta('AbacusSummit_base_c000_ph000', redshift=0.1)
print(meta['OmegaNow_m'])  # Omega_M(z=0.1)
print(meta['OmegaNow_DE'])  # Omega_DE(z=0.1)
```

6.1.2 Growth Factors in AbacusSummit

Growth factors in AbacusSummit are a bit of a special case, because the parameters actually contain a pre-computed table of $D(z)$ for all the output epochs and the ICs. This table is a `dict` called `GrowthTable`. Since AbacusSummit input power spectra (i.e. CLASS power spectra) are generated at $z = 1$, one can compute the linear power spectrum at a different epoch via the ratio $D(z)/D(1)$:

```
import abacusnbody.metadata
meta = abacusnbody.metadata.get_meta('AbacusSummit_base_c000_ph000')
Dz = meta['GrowthTable']
ztarget = 0.1
linear_pk = input_pk * (Dz[ztarget] / Dz[1.])**2
```

6.2 Developer Details

The metadata is stored in an ASDF file in the `abacusnbody/metadata` directory. The metadata files are built with `scripts/metadata/gather_metadata.py`, and then compressed with `scripts/metadata/compress.py` (using compression on the pickled representation). Internally, the time-independent parameters are separated from the time-varying state values, but the two sets are combined into a single `dict` that is passed to the user.

6.3 API

Retrieve the cosmology and other code parameters associated with an Abacus simulation.

Each set of simulations, like AbacusSummit, has a corresponding repository of metadata. The simulation name will be used to infer which repository to look in.

`abacusnbody.metadata.`**`get_meta`**(*simname*, *redshift=None*)

Get the metadata associated with the given simulation.

Parameters

- **`simname`** (*`str`*) – The simulation name, like "AbacusSummit_base_ph000_c000".
- **`redshift`** (*`float or str`*) – The redshift

Returns

meta – The time-independent parameters and, if *redshift* is given, the time-dependent state values.

Return type

dict

CHAPTER

SEVEN

ABACUSNBODY API

7.1 abacusnbody.data

7.1.1 compaso_halo_catalog module

See the *Python: compaso_halo_catalog* page.

7.1.2 pipe_asdf module

See the *Unix Pipes: pipe_asdf* page.

7.1.3 read_abacus module

This is an interface to read various Abacus file formats, like ASDF, pack9, and RVint.

For particle-oriented access to the data, one can use this interface. For halo-oriented access (e.g. associating halo particles with their host halo), one should use the relevant halo module (like `abacusnbody.data.compaso_halo_catalog`).

The decoding of the binary formats is generally contained in other modules (e.g. bitpacked); this interface mainly deals with the container formats and high-level logic of file names, Astropy tables, etc.

`abacusnbody.data.read_abacus.`**`read_asdf`**(*fn*, *load=None*, *colname=None*, *dtype=<class 'numpy.float32'>*, *verbose=True*, ***kwargs*)

Read an Abacus ASDF file. The result will be returned in an Astropy table.

Parameters

- **fn** (*str*) – The filename of the ASDF file to load
- **load** (*list of str or None, optional*) – A list of columns to load. The default (`None`) is to load columns based on what's in the file. If the file contains positions and velocities, those will be loaded; if it contains PIDs, those will be loaded.

  The list of fields that can be specified is: `'pos'`, `'vel'`, `'pid'`, `'lagr_pos'`, `'tagged'`, `'density'`, `'lagr_idx'`, `'aux'`

  All except `pos` & `vel` are PID-derived fields (see `abacusnbody.data.bitpacked.unpack_pids()`)
- **colname** (*str or None, optional*) – The internal column name in the ASDF file to load. Probably one of `'rvint'`, `'packedpid'`, `'pid'`, or `'pack9'`. In most cases, the name can be automatically detected, which is the default behavior (`None`).

- **dtype** (*np.dtype, optional*) – The precision in which to unpack any floating point arrays. Default: np.float32
- **verbose** (*bool, optional*) – Print informational messages. Default: True

Returns

table – A table whose columns contain the particle data from the ASDF file. The `meta` field of the table contains the header.

Return type

astropy.Table

7.1.4 bitpacked module

A collection of routines related to various Abacus bitpacked formats, like RVint and encoding of information in the PIDs.

Most users will not use this module directly, but will instead use *abacusnbody.data.compaso_halo_catalog* or *abacusnbody.data.read_abacus.read_asdf()*.

abacusnbody.data.bitpacked.**unpack_rvint**(*intdata*, *boxsize*, *float_dtype=<class 'numpy.float32'>*, *posout=None*, *velout=None*)

Unpack rvint data into pos and vel.

Parameters

- **intdata** (*ndarray of dtype np.int32*) – The rvint data
- **boxsize** (*float*) – The box size, used to scale the positions
- **float_dtype** (*np.dtype, optional*) – The precision in which to store the unpacked values. Default: np.float32
- **posout** (*ndarray, None, or False; optional*) – The array in which to store the unpacked positions. *None* can be given (the default), in which case an array is constructed and retured as the first return value. *False* can be given, in which case the positions are not unpacked.
- **velout** (*optional*) – Same as posout, but for the velocities

Returns

pos,vel – A tuple of the unpacked position and velocity arrays, or the number of unpacked particles if an output array was given.

Return type

tuple

abacusnbody.data.bitpacked.**unpack_pids**(*packed*, *box=None*, *ppd=None*, *pid=False*, *lagr_pos=False*, *tagged=False*, *density=False*, *lagr_idx=False*, *float_dtype=<class 'numpy.float32'>*)

Extract fields from bit-packed PIDs. The PID (really, the 64-bit aux field) enocdes the particle ID, the Lagrangian index (and therefore position), the density, and the L2 tagged field.

Parameters

- **packed** (*array-like of np.uint64, shape (N,)*) – The bit-packed PID (i.e. the aux field)
- **box** (*float, optional*) – The box size, needed only for `lagr_pos`
- **ppd** (*int, optional*) – The particles-per-dimension, needed only for `lagr_pos`
- **pid** (*bool, optional*) – Whether to unpack and return the unique particle ID.

  Loaded as a `np.int64` array of shape *(N,)*.

- **lagr_idx** (*bool, optional*) – Whether to unpack and return the Lagrangian index, which is the *(i,j,k)* integer coordinates of the particle in the cubic lattice used in Abacus pre-initial conditions. The `lagr_pos` field will automatically convert this index to a position.

  Loaded as a `np.int16` array of shape *(N,3)*.

- **lagr_pos** (*bool, optional*) – Whether to unpack and return the Lagrangian position of the particles, based on the Lagrangian index (`lagr_idx`).

  Loaded as array of type `dtype` and shape *(N,3)*.

- **tagged** (*bool, optional*) – Whether to unpack and return the CompaSO L2 tagged bit of the particles— whether the particle was ever part of an L2 group (i.e. halo core).

  Loaded as a `np.bool8` array of shape *(N,)*.

- **density** (*bool, optional*) – Whether to unpack and return the local density estimate, in units of mean density.

  Loaded as a array of type `dtype` and shape *(N,)*.

- **float_dtype** (*np.dtype, optional*) – The dtype in which to store float arrays. Default: `np.float32`

Returns

unpacked_arrays – A dictionary of all fields that were unpacked

Return type

dict of ndarray

7.1.5 asdf module

This module contains the ASDF extensions that allow the asdf Python package to read Abacus ASDF files that use Blosc compression internally.

There are two classes here: an Extension subclass, and a Compressor subclass. The Extension is registered with ASDF via a setuptools "entry point" in setup.py. It contains the reference to the Compressor subclass that knows how to handle Blosc compression.

abacusnbody.data.asdf.**set_nthreads**(*nthreads*)

class abacusnbody.data.asdf.**BloscCompressor**

Bases: `Compressor`

An implementation of Blosc compression, as used by Abacus.

property label

The string labels in the binary block headers that indicate Blosc compression

compress(*data*, ***kwargs*)

Useful compression kwargs: nthreads compression_block_size blosc_block_size shuffle typesize cname clevel

decompress(*blocks*, *out*, ***kwargs*)

Useful decompression kwargs: nthreads

class abacusnbody.data.asdf.**AbacusExtension**

Bases: `Extension`

An ASDF Extension that deals with Abacus types and formats. Currently only implements Blosc compression.

property extension_uri

Get the URI of the extension to the ASDF Standard implemented by this class. Note that this may not uniquely identify the class itself.

Return type

str

property compressors

Return the Compressors implemented in this extension

7.1.6 pack9 module

Unpack pack9 particle data, which encodes the pos + vel in 9 bytes, and then the PID + aux in another 8 bytes in a separate file. The 9-byte part is handled by this module; the 8-byte (PID) part is handled by `bitpacked.unpack_pids()`.

Most users will not use this module directly, but will instead use the *abacusnbody.data.read_abacus.read_asdf()* function.

abacusnbody.data.pack9.**unpack_pack9**(*data*, *boxsize*, *velzspace_to_kms*, *float_dtype=<class 'numpy.float32'>*, *posout=None*, *velout=None*)

7.2 abacusnbody.hod

7.2.1 abacus_hod module

See the *AbacusHOD* page.

CHAPTER

EIGHT

CHANGELOG

8.1 1.3.0 (2022-06-08)

8.1.1 Breaking Changes

- Dependencies for tests and scripts now factorized under `abacusutils[test]` and `abacusutils[extra]` [#46]
- Python 3.6 (EOL) support has been dropped [#56]

8.1.2 New Features

- `abacusnbody.metadata` added. Supports querying simulation parameters without downloading simulation data. [#56]

8.1.3 Fixes

- Fix periodicity in theory-box HOD, and add halo LC features [#41]
- Fix read lc rv [#37]

8.1.4 Enhancements

- Some nice numba accelerations for fenv calculation [#45]
- Made clustering_params optional, among some minor quality of life updates. [#39]
- Reduce memory usage in Menv tree queries [#51]
- HOD now supports two new conformity parameters for ELGs, conf_a, conf_c [#54]

8.2 1.2.0 (2022-02-02)

8.2.1 New Features

- Now supports Python 3.10 [#19]
- HOD module now works with halo light cone catalogs [#28]

8.3 1.1.0 (2022-01-21)

8.3.1 Fixes

- Fixed issues with QSO incompleteness [#15]
- Fix `cleandir` and propagate cleaning info in header [#18]

8.3.2 New Features

- Add `filter_func` superslab filtering to `CompaSOHaloCatalog` [#16]
- Add pack9 reader [#25]
- Add light cone catalog reading to `CompaSOHaloCatalog` [#11]

8.3.3 Enhancements

- Sped up RNG for reseeding [#24]

8.3.4 Changes

- Migrate testing to GitHub CI; start some linting [#17]
- Automatic versioning and releasing [#27]

8.4 1.0.4 (2021-07-15)

8.4.1 Fixes

- Fix IC parameter in config file and ELG HOD generation

8.5 1.0.3 (2021-06-16)

8.5.1 Fixes

- Fix HOD `prepare_sim` error when `want_AB = False` [#14]

8.5.2 Changes

- Start testing Python 3.9 [#13]

8.6 1.0.2 (2021-06-04)

8.6.1 Changes

- Relax numba version requirement for DESI Conda compatibility. Warning: `numba<0.52` not fully tested with `abacusnbody.hod` package.

8.7 1.0.1 (2021-06-03)

8.7.1 Changes

- Use updated directory structure for cleaned catalogs.

8.8 1.0.0 (2021-06-02)

8.8.1 Fixes

- Fixed issue where satellite galaxy halo ID was incorrect.

8.8.2 New Features

- `CompaSOHaloCatalog` can read "cleaned" halo catalogs with `cleaned=True` (the default) [#6]

8.8.3 Breaking Changes

- Can no longer load field particles or L0 halo particles through `CompaSOHaloCatalog`; use `abacusnbody.data.read_abacus.read_asdf()` to read the particle files directly instead. [#6]

8.8.4 Enhancements

- AbacusHOD now supports cleaned catalogs and uses them by default [#6]
- Printing a `CompaSOHaloCatalog` now shows the memory usage (also available with `CompaSOHaloCatalog.nbytes()`) [#6]
- Our custom fork of ASDF is no longer required [#10]

8.8.5 Deprecations

- Passing a string to the `load_subsamples` argument of `CompaSOHaloCatalog` is deprecated; use a dict instead, like: `load_subsamples=dict(A=True, rv=True)`. [#6]
- `cleaned_halos` renamed to `cleaned`

8.9 0.4.0 (2021-02-03)

8.9.1 New Features

- Add `AbacusHOD` module for fast HOD generation using AbacusSummit simulations [#4]
- `CompaSOHaloCatalog` constructor now takes field names in the `unpack_bits` field

8.9.2 Enhancements

- Bump minimum Blosc version to support zero-copy decompression in our ASDF fork

8.10 0.3.0 (2020-08-11)

8.10.1 Enhancements

- Use 4 Blosc threads for decompression by default

8.10.2 Fixes

- Specify minimum Astropy version to avoid `AttributeError: 'numpy.ndarray' object has no attribute 'info'`

8.11 0.2.0 (2020-07-08)

8.11.1 New Features

- Add pipe_asdf.py script as an example of using Python to deal with file container so that C/Fortran/etc don't have to know about ASDF or blosc

8.12 0.1.0 (2020-06-24)

8.12.1 New Features

- CompaSOHaloCatalog accepts `fields` keyword to limit the IO and unpacking to the requsted halo catalog columns

8.13 0.0.5 (2020-05-26)

- First stable release

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

E

G

I

J

L

M

N

R

S

U