CAESAR Documentation

Release 0.1

Robert Thompson

Dec 26, 2021

CONTENTS

CAESAR is a python-based yt extension package for analyzing the outputs from cosmological simulations. CAESAR takes as input a single snapshot from a simulation, and outputs a portable and compact HDF5 catalog containing a host of galaxy and halo properties that can be read in and explored without the original simulation binary. CAESAR thus provides a simple and intuitive interface for exploring object data within your outputs.

CAESAR provides further functionality such as identifying the most massive progenitors or descendants across snapshots (see Progenitors), and generating FSPS photometry and spectra for galaxies (see Photometry). Also, the CAESAR catalog contains particle ID lists for each galaxy/halo, enabling you to quickly grab the relevant particle data in the original snapshot in order to compute any other galaxy/halo quantity you want.

CAESAR is OpenMP-parallelized using cython-parallel and joblib. It enjoys decent scaling with the (user-specifiable) number of cores. Catalog generation does, however, have substantial memory requirements – e.g. a run with two billion particles requires a machine with 512 GB to generate the catalog, and this scales with the number of particles. The resulting CAESAR catalog typically has a filesize of less than 1% of the original snapshot, so once this is generated, using it is not memory-intensive.

CAESAR generates a catalog as follows:

1. Identify halos (or import a halo membership list)
2. Compute halo physical properties
3. Within each halo, identify galaxies using 6-D friends-of-friends
4. Compute galaxy physical properties
5. Optionally, compute galaxy photometry including line-of-sight extinction
6. Create particle lists for each galaxy and halo
7. Link galaxies and halos, identify centrals+satellites, quantify environment
8. Output all information into a stand-alone hdf5 file

Once the CAESAR catalog has been generated, it can be loaded and the data easily accessed using simple python commands.

CAESAR builds upon the yt project, which provides support for a number of simulation codes and symbolic units. All meaningful quantities stored within a CAESAR catalog have units attached, reducing ambiguity when working with your data. This tight connection enables you to use both yt and CAESAR functionality straightforwardly within a single analysis package.

CAESAR currently supports the following codes/formats:

1. GADGET

2. GIZMO
3. TIPSY
4. ENZO
5. ART
6. RAMSES

In principle, any `yt`-supported simulation snapshot could be supported by `CAESAR`, but some aspects may not work out-of-the-box owing to different conventions for e.g. metallicity arrays. `CAESAR` has been tested on Mufasa, Simba, Illustris/TNG, and EAGLE snapshots. We happily accept pull requests for further functionality and bug fixes.

To get started, follow the `Getting Started` link below!

CHAPTER

ONE

CONTENTS

1.1 Getting Started

1.1.1 Requirements

- python >= 3.x
  - numpy
  - scipy
  - cython
  - h5py
  - matplotlib
  - psutil
  - joblib
  - six
  - astropy
  - yt >= 4.0

1.1.2 Installation

Python and friends

Since this is a python package, you must have python installed! `CAESAR` formally requires python-3. Some basic functionality is still compatible with python-2, but we have discontinued further support for this in `CAESAR`.

We strongly encourage using a pre-packaged python distribution, such as Anaconda. This will install an isolated python environment in your home directory giving you full access to install and change packages without fear of screwing up your system's default python install. Another advantage is that it comes with nearly everything you need to get started working with python (numpy/scipy/matplotlib/etc).

Dependencies

Installing the main dependencies is very easy under Anaconda, or using the python package manager pip.

```
$> conda install numpy scipy cython h5py matplotlib psutil joblib six astropy
```

Alternatively, if you do not wish to use Anaconda, these can all be installed under `pip` by replacing `conda` with `pip` in the line above. Some of these automatically come with Anaconda, but the above command will update these to the latest version if needed.

Be aware that in order for `h5py` to properly compile you must first have HDF5 correctly installed (via e.g. *apt-get*, *brew*, or manual compilation) and in your respective environment paths.

The optional galaxy/halo photometry computation in `CAESAR` requires python-fsps, which is a python wrapper for the `FSPS` fortran package. Please follow their installation instructions to install this. Furthermore, you will also need two other packages that are only available via `pip`:

```
$> pip install synphot extinction
```

If you wish to use the MPI driver to run single instances of Caesar over many cores via MPI, it is also necessary to install `mpi4py`:

```
$> conda install mpi4py
```

Note that `CAESAR` is natively OpenMP-parallel, and the MPI implementation may be system-specific.

If you wish to work with galaxy and halo particle lists (for instance to compute your own quantities) it is highly recommended that you install `pygadgetreader`:

```
$> git clone https://github.com/dnarayanan/pygadgetreader.git
$> cd pygadgetreader
$> python setup.py install
```

---

yt

CAESAR builds on the yt simulation analysis toolkit. CAESAR requires yt version >=4.0, though a lot of functionality will still work with yt-3.6+.

We recommend installing yt via Anaconda:

```
$> conda install -c conda-forge yt
```

but other installation options are described here.

If you already have yt, you can check your version using `yt version`, and update if necessary.

CAESAR

Now that we have all of the prerequisites out of the way we can clone and install CAESAR:

```
$> git clone https://github.com/dnarayanan/caesar.git
$> cd caesar
$> python setup.py install
```

Once it finishes you should be ready to finally get some work done!

1.1.3 Updating

To update CAESAR simply pull the changes and reinstall:

```
$> cd caesar
$> git pull
$> python setup.py install
```

1.2 Running CAESAR

CAESAR offers three basic functions:

[1] Identify galaxies and halos, compute a wide range of properties for each object, and cross-match them.

[2] Compute photometry accounting for the line-of-sight dust extinction to each star in the object.

[3] Compute the N most massive progenitors/descendants for any galaxy or halo in another snapshot (see Progenitors docs page for usage).

1.2.1 Scripted

It is generally recommended to run CAESAR within a script for computing galaxy and halo properties. This allows for more precise control over the various options, looping over many files, parallelizing, etc. Here is a basic script for running member_search:

```
import yt
import caesar

# first we load your snapshot into yt
ds = yt.load('my_snapshot')

# now we create a CAESAR object, and pass along the yt dataset
obj = caesar.CAESAR(ds)

# now we execute member_search(), which identifies halos, galaxies, and computes
# properties (including photometry; this requires installing FSPS) on 16 OpenMP cores
obj.member_search(haloid='fof',fof6d_file='my_fof6dfile',fsps_bands='uvoir',ssp_model=
↪'FSPS',ssp_table_file='FSPS_Chab_EL.hdf5',ext_law='composite',nproc=16)

# finally we save the CAESAR galaxy/halo catalog to your desired filename
obj.save('my_caesar_file.hdf5')
```

member_search() options

Here is a more detailed description of the options shown above:

- **nproc**: Number of cores for OpenMP parallelization. This follows the `joblib` convention that negative numbers correspond to using all except `nproc+1` cores, e.g. `nproc=-2` uses all but 1 cores. *Default:* 1
- **haloid**: Source for particle halo ID's. `haloid='fof'` uses a 3D Friends-of-Friends (3DFOF) with b=0.2 to identify halos. `haloid='snap'` reads halo membership info for each particle from the snapshot variable `HaloID`, if present. *Default:* 'fof'
- **fof6d_file**: Stores results of 6DFOF galaxy finder in a file for future retrieval. If file does not exist, it is created; if it exists, the galaxy membership information is read from this file instead of running the 6DFOF. *Default: None*
- **fsps_bands**: Triggers optional photometry computation, in specified list of bands. The `fsps.list_filters()` command under `python-fsps` lists the available bands. One can also specify a string (minimum 4 characters) that will be matched to all available bands, e.g. `fsps_bands=['sdss','jwst']` will compute all bands that include the phrase `sdss` or `jwst`. *Default: None*
- **ssp_model**: Choice of `FSPS`, `BPASS`, or `BC03` (Bruzual-Charlot 2003). *Default: None*
- **ssp_table_file**: Path to lookup table for FSPS photometry. If this file does not exist or this keyword is unspecified, it will be generated; this takes some time. If it exists, CAESAR will read it in. *Default: None*

1.2.2 Command Line

NOTE: CURRENTLY, RUNNING FROM THE COMMAND-LINE IS NOT OPERATIONAL. Please use the Scripted method described above.

1.3 Loading CAESAR files

1.3.1 Command Line

Using the CLI we can load our CAESAR file from the previous example automatically and have it drop us at an `ipython` prompt via:

```
$> caesar caesar_snapshot.hdf5
```

This will open up the `caesar_snapshot.hdf5` file and check for the `CAESAR=True` attribute in the HDF5 header. If found it will proceed to deserialize the `CAESAR` file and drop you to an interactive python prompt with access to the *main.CAESAR* object via the `obj` variable. At this point you are free to explore the data structure and manipulate at will.

1.3.2 Scripted

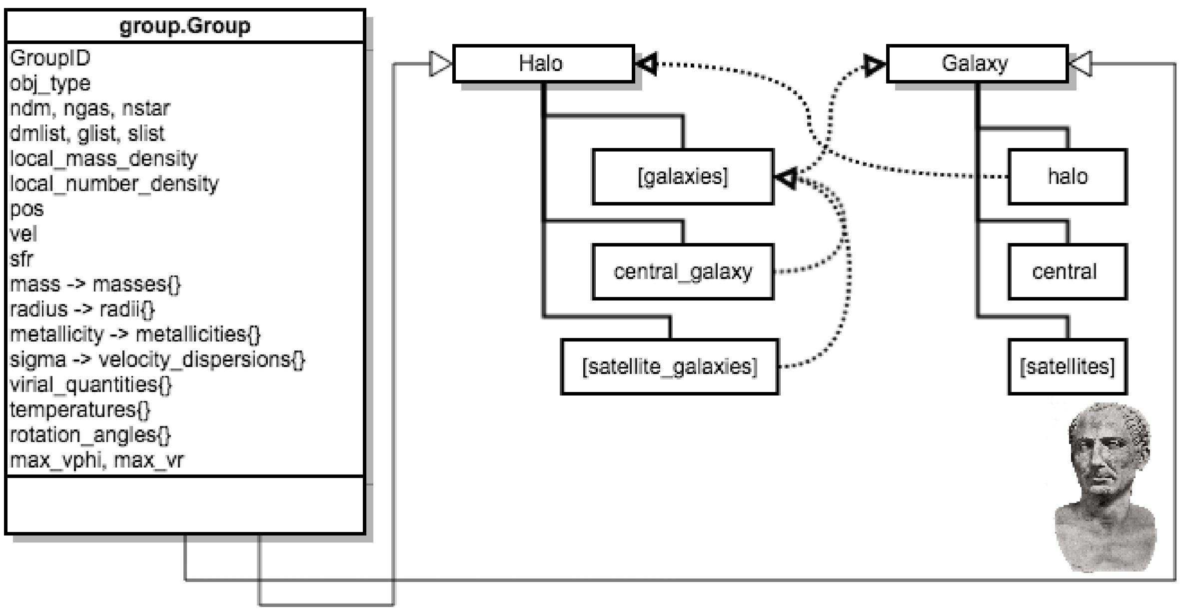
In order to do more in depth analysis, you will likely want to built your own analysis scripts. Before getting into the nuts and bolts of your analysis you will need to load in your `CAESAR` file to gain access to all objects and their respective attributes. This can be accomplished with the following code:

```python
import caesar

# define input file
infile = 'caesar_snapshot.hdf5'

# load in input file
obj = caesar.load(infile)
```

1.4 Using CAESAR

1.4.1 Data Structure

Within the `main.CAESAR` object you will find a number of *lists* containing any number of `group.Group` objects. The primary lists are the `halos` and `galaxies` list; within each of these you will find every object identified by `CAESAR`. Below is a quick relationship diagram describing how objects are linked together:



From this you can see that each `Halo` object has a list of galaxies, a link to its central galaxy, and a sub-list of satellite galaxies (those who are not a central). Each `Galaxy` object has a link to its parent `Halo`, a boolean describing if it is a central, and a sub-list linking to all of the satellite galaxies for its parent halo.

1.4.2 Usage

Usage of `CAESAR` all comes down to *what* you want to do. The real power of the code comes with the simple relationships that link each object. As we saw in the previous section, each `group.Group` has a set of relationships. We can exploit these to intuitively query our data. For example, say you wanted to get an array of all galaxy masses, how would you most efficiently do that? The easiest way (in my opinion) would be to use python's list comprehension. Here is a quick example (assuming you have the `main.CAESAR` object loaded into the `obj` variable):

```
galaxy_masses = [i.masses['total'] for i in obj.galaxies]
```

This is basically a compact way of writing:

```
galaxy_masses = []
for gal in obj.galaxies:
    galaxy_masses.append(gal.masses['total'])
```

Now that in itself is not all that impressive. Things get a bit more interesting when we start exploiting the object relationships. As another example, say we wanted the *parent halo mass* of each galaxy? Lets see how that is done:

```
parent_halo_masses = [i.halo.masses['total'] for i in obj.galaxies]
```

Since each `group.Galaxy` has a link to its parent `group.Halo`, we have access to all of the parent halo's attributes. We can also begin to add conditional statements to our list comprehension statements to further refine our results; let's only look at the halo masses of massive galaxies:

```
central_galaxy_halo_masses = [i.halo.masses['total'] for i in obj.galaxies if i.masses[
↪'total'] > 1.0e12]
```

Obviously we can make these list comprehensions infinitely complicated, but I think you get the gist. The bottom line is: **CAESAR provides a convenient and intuitive way to relate objects to one another**.

1.5 Catalog Quantities

CAESAR computes many quantities for galaxies (from 6DFoF) and halos identified within a given simulation snapshot. Below we describe the structure of the catalog file, and the quantities that are computed.

1.5.1 Structure

The top level of the catalog `hdf5` file contains:

```
$> h5ls CAESARFILE
galaxy_data              Group
global_lists             Group
halo_data                Group
simulation_attributes    Group
tree_data                Group
```

`galaxy_data` and `halo_data` contain the galaxy and halo catalogs, respectively. `simulation_attributes` contains various simulation parameters. `global_lists` contains some auxiliary lists; there is no good reason to directly access this. Finally, `tree_data` contains the output of running `progen`, which is not initially created by `CAESAR` but can be added later (click on *Progenitors* tab for more info).

1.5.2 galaxy_data

`galaxy_data` contains some computed quantities for each galaxy, but many of the quantities are in *dictionaries* which are listed in `dicts` (described below). There is also `lists`, which stores particle lists, but it shouldn't be necessary to directly access this.

The list of quantities can be seen using `h5ls`:

```
$> h5ls CAESARFILE/galaxy_data
```

Quantities stored at the top level in `galaxy_data` are:

- `GroupID` – A sequential ID number for each galaxy
- `parent_halo_index` – Index number in `halo_data` for the galaxy's parent halo
- `central` – Flag to indicate whether galaxy is central (i.e. most massive in stars) within its halo (1), or a satellite (0)
- `pos,vel` – Center-of-mass (CoM) position and velocity, including units (typically kpccm)
- `ngas,nstar,ndm,nbh` – Number of particles of each particle type
- `sfr` – Instantaneous star formation rate in Msun/yr, from summing SFR in gas particles
- `sfr_100` – SFR averaged over last 100 Myr, from star particles formed in that time.
- `bhmdot`, `bh_fedd` – Central black hole accretion rate in Msun/yr, and central BH eddington ratio. The central black hole is taken to be the most massive one, if there are multiple BH in the galaxy.
- `L_FIR` – If photometry was done, this will contain the bolometric far-IR luminosity in erg/s (i.e. the total energy absorbed by dust extinction)
- Various list start/end values – These are indexes for the particle lists; these should not be accessed directly, but rather through `glist`, `slist`, etc. (see below)

1.5.3 Dictionary quantities

`dicts` contains the majority of the computed quantities. These are accessed via a dictionary key, e.g. `obj.galaxies[0].masses['stellar']` gives the stellar mass of the first galaxy in the catalog. The full list of quantities in any given file can be seen using:

```
$> h5ls CAESARFILE/galaxy_data/dicts
```

`dicts` contains the following quantities:

- `masses`: `['gas','stellar','dm','dust','bh','HI','H2']` as well as many corresponding quantities within 30 kpccm spherical apertures denoted by `_30kpc` attached to each name. The first 5 of these come directly by summing particle masses. The `HI` and `H2` masses come from assigning all the gas in the halo to its most bound galaxy within the halo (see Dave et al. 2020). Note that the `dm` mass from 6DFOF is 0 by definition, since the 6DFOF does not consider DM particles; the `dm_30kpc` however will be nonzero.
- `radii`: `['gas_XX','stellar_XX','dm_XX','bh_XX','baryon_XX', 'total_XX']`, where `XX` is `half_mass`, `r20`, or `r80`, which are the radii enclosing 50, 20, and 80 percent of the mass of the given type. The galaxy center of mass from which the radii are found is recomputed for each type. `baryon` includes `gas`, `stellar`, and `bh`, while `total` includes `dm` as well.

- `metallicities`: `['mass_weighted','sfr_weighted','stellar','mass_weighted_cgm', 'temp_weighted_cgm']`. The first two are gas-phase, weighted as indicated. The stellar metallicity is mass-weighted. The CGM metallicities are for gas outside galaxies (*n_H<0.13* H atom/cm^3); this is only meaningful for halos. These are in total metal mass fractions (not solar-scaled). This uses Metallicity[0] from the snapshot, which is the total metallicity; `CAESAR` does not have any information regarding specific elements, this must be obtained from the snapshot directly if desired using e.g. `pygadgetreader`.
- `velocity_dispersions`: `['gas','stellar','dm','bh','baryon', 'total']`. Mass-weighted velocity dispersions for each particle type, computed around the CoM velocity (recomputed for each type). These are in km/s.
- `rotation`: `['gas_XX','stellar_XX','dm_XX','bh_XX','baryon_XX', 'total_XX']`, where `XX` here can be `L`, `ALPHA`, `BETA`, `BoverT`, and `kappa_rot`. `L` (3 components) is the angular momentum vector of the galaxy in Msun-kpccm-km/s. `ALPHA` and `BETA` are rotation angles required to rotate the galaxy to align with the angular momentum. `BoverT` is bulge-to-total mass ratio, where the bulge mass is defined kinematically as twice the counter-rotating mass. `kappa_rot` is the fraction of kinetic energy in rotation, as defined in Sales et al. (2012).
- `ages`: `['mass_weighted','metal_weighted']` Mean stellar ages, weighted by mass or (additionally) metallicity.
- `temperatures`: `['mass_weighted','mass_weighted_cgm','temp_weighted_cgm']` These are the average temperatures of the gas within galaxies or in the CGM. Owing to the assumed equation of state in cosmological simulations, this is typically not very meaningful for galaxies. However, it is useful for halos.
- `local_mass_density` and `local_number_density`: `[300,1000,3000]`. Environmental measures giving the mass and number density of `CAESAR` galaxies within spherical tophat apertures as indicated in kpccm.
- Photometry: `absmag` and `appmag`, along with corresponding `_nodust` values, for all the photometric bands computed (if photometry was run). More information is available in the Photometry docs.

1.5.4 halo data

`halo_data` contains many of the same quantities as `galaxy_data`. However, there are some crucial differences.

At the top level, there are some new quantities:

- `minpotpos`, `minpotvel`: Position and velocity of the particle with the lowest potential in the halo. This is often a more useful that the CoM values within halos, since FoF halos can be quite irregular in shape.
- `central_galaxy`: GroupID of central galaxy in the halo.
- `galaxy_index_list_start/end`: This is the indexing for the list of galaxy GroupID's in the halo. DO NOT USE THESE VALUES DIRECTLY TO LOOK IN `galaxy_data`! These are cross-indexed, so to get the galaxy indexes within a given halo use

```
In[1]: halogals = np.asarray([i.galaxy_index_list for i in obj.halos])
```

Meanwhile, in `halo_data/dicts`, beyond all the `galaxy_data` dictionaries (except photometry) there is a new dictionary called `virial_quantities`:

- `virial_quantities`: `['circular_velocity','spin_param','temperature','mXXXc', 'rXXXc']`: Circular velocity=*sqrt(GM_tot/R_tot)* where *R_tot* is the equivalent radius that would enclose *M_tot* at an overdensity of 200 times the critical. The `XXX` quantities for mass and radii are computed within `200`, `500`, or `2500` times the critical density, by expanding a sphere around `minpotpos` until the mean density within drops below that value. Note that only halo particles are included, so owing to the irregular shapes of FoF halos, this can lead to `200` quantities sometimes missing significant mass; for `500` and `2500` the effects are quite small. Overall, these values should be regarded as somewhat approximate to be used for rough analyses.

1.5.5 Particle lists

Each halo and galaxy contains a list of particles indexed by particle type. For gas, stars, DM, and BHs these are `glist`, `slist`, `dmlist`, and `bhlist`, respectively. These lists contain the indexes of particles of a given type within the original snapshot. These lists allow the user to compute any desired quantity, by looking up the required quantities within the original snapshot.

To use these lists, one must read in the particles from the snapshot. This can be done for instance using `pygadgetreader`. For instance, the `CAESAR` file does not contain metallicities of individual elements. So one might desire, e.g. the SFR-weighted oxygen abundance.

To do this, we first use `pygadgetreader` to read in the particle lists:

```
In[1]: import caesar
In[2]: from readgadget import readsnap  # pygadgetreader
In[3]: obj = caesar.load(CAESARFILE)
In;4]: h = obj.simulation.hubble_constant  # H0/100
In[5]: gsfr = readsnap(SNAPFILE,'sfr','gas',units=1) # particle SFRs in Mo/yr
In[6]: gmetarray = readsnap(SNAPFILE,'Metallicity','gas') # For Simba, this is an 11-
↪element array per particle
In[7]: pOgas = np.asarray([i[4] for i in gmetarray])  # For Simba, oxygen is 5th element␣
↪in the Metallicity array
```

Next, we use `glist` to compile the particles in each galaxy, and use them to compute the SFR-weighted oxygen abundance:

```
In[8]: Zoxy = []
In[9]: for g in sim.galaxies:
In[10]:     psfr = np.array([gsfr[k] for k in g.glist]) # particle sfr's
In[11]:     ZO = np.array([pOgas[k] for k in g.glist]) # oxygen mass fraction
In[12]:     Zoxy.append(np.sum(ZO*psfr)/np.sum(psfr))
```

This fills an array `Zoxy` with the SFR-weighted metallicity.

In this way, `CAESAR` (plus a particle reader of your choice) enables the computation of any quantity associated with a given galaxy or halo object.

1.6 Progenitors

The `progen` module in `CAESAR` links groups across snapshots, by computing the most massive progenitor(s) or descendant(s) for each group in a different snapshot. Groups (i.e. `galaxy/halo/cloud`) are linked by finding the most particles in common of a specified particle type (e.g. `star`). If snapshot numbers are specified in falling order, then progenitors are computed; if in rising order, then descendants are computed. The information is appended into the

CAESAR file within the hdf5 dataset tree_data, and are stored separately for progenitors and descendants, as well as separately for each group type and particle type.

1.6.1 Progen over many snapshots

run_progen() is the simplest way to run progen over a list of snapshots, e.g.:

```
In [1]: caesar.progen.run_progen('/path/to/snapshots/for/m25n256', 'snap_m25n256_',␣
↪list(range(151,0,-1), prefix='caesar_')
```

This will find progenitors (since the snapshots are specified in falling order) in snaphots 0-151 for the snapshots in the directory provided as the first argument, with the snapshot basename provided as the second argument. Any snapshots for which a snapshot file or Caesar file are not found, or for which there is no halo_data, are ignored (with a warning).

The snapshot are linked via daisychaining. That is, in the example above, 151 is linked to 150, 150 to 149, and so on (assuming they all exist). If you want to link two particular snapshots, see "Linking two specific snapshots".

The prefix option specifies the name prefix for the corresponding CAESAR file in the Groups subdirectory; in this case, snap_m25n256_151.hdf5 should have its CAESAR file in Groups/caesar_m25n256_151.hdf5, etc. The example above uses default options for linking progenitors/descendants; other choices can be specified as noted in "Progen options" below. run_progen() only writes the information to the CAESAR file, it does not return anything.

1.6.2 Linking two specific snapshots

progen_finder() links the groups in two specified CAESAR objects, and then writes it to the specified CAESAR file. While normally called from run_progen(), it can be run stand-alone as well. This is useful if e.g. your locations for snapshots and Caesar files are not as assumed in run_progen(). Here is an example using progen_finder():

```
In [1]: import caesar
In [2]: obj1 = caesar.load(caesarfile1)
In [3]: obj2 = caesar.load(caesarfile2)
In [4]: my_progens = caesar.progen.progen_finder(obj1, obj2, caesarfile1)
```

plus any options you desire as listed in "Progen options".

progen_finder() returns the progenitor or descendant list, as well as (by default) writing to the CAESAR file. If you specify overwrite=False, the progenitor/descendant list is returned without actually writing anything to the Caesar file. This is useful if you want to link two particular snapshots but don't want to save that for posterity.

1.6.3 Progen options

The following options can be passed to run_progen() or progen_finder():

- data_type: Group type to find progen/descend info for; can be galaxy, halo, or cloud. *Default:* galaxy
- part_type: Particle type to find progen/descend info for. *Default:* star
- n_most: Finds the n_most most massive progenitors/descendants. If n_most>1, the info is then stored in a array of size (ngroups,n_most). Currently can only be 1 or 2. *Default:* 1
- min_in_common: Requires that the current group and the prog/desc group have at least this fraction of particles in common to be considered valid. *Default:* 0.1
- overwrite: If True, (over)writes info into CAESAR file. If False, then if it already exists read it in and return it; but if it doesn't already exist, compute and return it but don't touch the CAESAR file. *Default:* True

- `nproc`: Number of OpenMP cores (using joblib, passed as `n_jobs`). `progen` is already very fast, so this isn't terribly useful, except maybe for DM halos where there are lots of groups and particles. *Default:* 1

1.6.4 Where is the info stored?

By default, the progenitor/descendant info is stored in the `tree_data` dataset within the `CAESAR` file. This is a separate dataset from `galaxy_data`, `halo_data`, etc. Within this, the information is stored as numpy arrays of integers, where each integer corresponds to the index of the group in the other snapshot that is its progenitor/descendant info.

The index name for each array is created by concatenating three pieces of information: Whether it is a progenitor or descendant; the group type; and the particle type. So an example might be `progen_galaxy_star`, meaning that the indexes in that array are progenitors of galaxies linked via most numbers of stars in common. This array will have exactly as many entries as there are galaxies in `galaxy_data`.

Each of 3 group types can be linked in two ways (`progen/descend`) via each of 6 particle types, making for 36 potential index names being stored in `tree_data`. In detail, galaxies and clouds do not include dark matter particles so e.g. `descend_galaxy_dm` or `progen_cloud_dm2` cannot exist, so there are actually 28 potential index names.

Additionally, `tree_data` hold the redshift for which the progenitors and/or descendants have been identified. You can retrieve this info using the `get_progen_redshift()` command:

```
In [1]: redshift = caesar.progen.get_progen_redshift(my_caesar_file,'descend_galaxy_star
↪')
```

or similarly for any other choice of `index_name`.

1.6.5 Auxiliary routines

Some other potentially useful routines are available in `progen`:

- `z_to_snap(redshift, snaplist_file, mode)` finds the closest snapshot in redshift to the provided redshift, from the list specified in `snaplist_file`. Specifying `snaplist_file=Simba` uses the snapshot values in the Simba simulation suite. Returns the snapshot number and its redshift.
- `wipe_progen_info(caesar_file, [index_name])` removes `index_name` info from `caesar_file`. With no `index_name` (default), it wipes all datasets containing the word `progen` or `descend`; this should return the `CAESAR` file to the state before any progen was run.
- `check_if_progen_is_present(caesar_file, index_name)` checks if the dataset `index_name` is in the `CAESAR` file `caesar_file`
- `collect_group_IDs(obj, data_type, part_type, snap_dir)` collects all groups IDs for a given `data_type` and `part_type` into a single array, and returns the particle and group IDs along with a hash array of length `ngroups` which marks the locations of the start of each group.

1.7 Photometry

CAESAR can optionally compute photometry for any object(s) in any available FSPS band. This is done as in Pyloser: Compute the dust extinction to each star based on the line-of-sight dust column, attenuate its spectrum with a user-selectable attenuation law, sum the spectra of all stars in the object, and apply the desired bandpasses.

NOTE: CAESAR accounts for dust but does *not* do proper dust radiative transfer! To e.g. model the far-IR spectrum or predict extinction laws, you can use Powderday. The main advantage of CAESAR is speed. Also, it gives the user more direct control over the attenuation law used, which may be desirable in some instances. Results are similar to Powderday for most galaxies, but differences at the level of ~0.1 magnitudes are not uncommon.

1.7.1 Installation

To compute photometry, two additional packages must be installed:

- python-fsps: Follow the instructions, which requires installing and compiling FSPS.
- synphot: Available via `pip` or in `conda-forge`.

1.7.2 Running in member_search

The photometry computation for galaxies can be conveniently done as part of `member_search()`. This is invoked by specifying the `band_names` option to `member_search()`.

CAESAR will compute 4 magnitudes for each galaxy, corresponding to apparent and absolute magnitudes, with and without dust. These are stored in dictionaries `absmag`, `absmag_nodust`, `appmag`, and `appmag_nodust`, with keywords corresponding to each requested band (e.g. `absmag['sdss_r']`) When invoked within `member_search()`, CAESAR computes photometry for *all galaxies*. For doing halos/clouds/subset of galaxies, see *Running stand-alone* below.

For example, the following command will invoke `member_search` for a CAESAR object `obj`, which will do everything as before, then will additionally compute galaxy photometry for all SDSS and Hubble/WFC3 filters using an LMC extinction law viewed along the `z` axis:

```
In [1]: obj.member_search(band_names='[sdss,wfc3]',ssp_table_file='SSP_Chab_EL.hdf5',ext_
↪law='lmc',view_dir='z')
```

1.7.3 Running stand-alone

The photometry module can also be run stand-alone for specified objects. Any object with stars and gas (stored in `slist` and `glist`) can have its photometry computed. To do so, first create a photometry object, and then apply `run_pyloser()` to it.

For example, to run photometry for all halos in a pre-existing CAESAR catalog:

```
In [1]: from caesar.pyloser.pyloser import photometry
In [2]: ds  = yt.load(SNAP)
In [3]: sim = caesar.load('my_caesar_file.hdf5')
In [4]: galphot = photometry(sim,sim.halos,ds=ds,band_names='sdss',nproc=16)
In [5]: galphot.run_pyloser()
```

All options as listed under "Photometry Options" are passable to photometry. The computed SDSS photometry will be available in the usual dictionaries absmag, absmag_nodust, appmag, and appmag_nodust, for each halo.

1.7.4 Photometry Options

The following options can be passed to member_search() or when instantiating the photometry class:

- band_names: (REQUIRED): The list of band(s) to compute, selected from python-fsps (use fsps.list_filters() to see options). You can also specify a substring (min. 4 characters) to do all bands that contain that substring, e.g. 'sdss' will compute all available SDSS bands. The v band is always computed; the difference between the absmag and absmag_nodust gives A_V. There are two special options: 'all' computes all FSPS bands, while 'uvoir' computes all bands bluewards of 5 microns. *Default*: ['v']
- ssp_table_file: Filename containing FSPS spectra lookup table. If it doesn't exist, it is generated assuming a Chabrier IMF with nebular emission and saved to this filename for future use. If you prefer different FSPS options, first generate it using generate_SSP_table, and read it in here. *Default*: 'SSP_Chab_EL.hdf5'
- ext_law: Specifies the extinction law to use. Current options are calzetti, chevallard, conroy, cardelli (equivalently mw), smc, and lmc. There are two composite extinction laws available: mix_calz_mw uses mw for galaxies with specific star formation rate sSFR<0.1 Gyr^-1, calzetti for sSFR>1, and a linear combination in between. composite additionally adds a metallicity dependence, using mix_calz_mw for Z>Solar, smc for Z<0.1*Solar, and a linear combination in between. *Default*: 'composite'
- view_dir: Sets viewing direction for computing LOS extinction. Choices are x, y, z. *Default*: 'x'
- use_dust: If present, uses the particles' dust masses to compute the LOS extinction. Otherwise uses the metals, with an assumed dust-to-metals ratio of 0.4, reduced for sub-solar metallicities. *Default*: True
- use_cosmic_ext: Applies redshift-dependent Madau(1995) IGM attenuation to spectra. This is computed using synphot.etau_madau(). *Default*: True
- nproc: Number of cores to use. If -1, it tries to use the CAESAR object's value, or else defaults to 1. *Default*: -1

1.7.5 Generating a lookup table

If you don't want Caesar's default choices of Chabrier IMF and nebular emission with all other options set to the python-FSPS default, you will need to create a new table and specify it with ssp_table_file when instantiating photometry.

To create a new SSP lookup table, run generate_ssp_table with the desired FSPS options. For example:

```
In [1]: from caesar.pyloser.pyloser import generate_ssp_table
In [2]: generate_ssp_table('my_new_SSP_table.hdf5',Zsol=0.0134,oversample=[2,2],imf_
↪type=1,add_neb_emission=True,sfh=0,zcontinuous=1)
```

Options:

- oversample oversamples in [age,metallicity] by the specified factors from the native FSPS ranges, in order to get more accurate interpolation. Note that setting these >1 creates a larger output file, by the product of those values. *Default*: [2,2]
- Zsol sets the metallicity in solar units in order to convert the FSPS metallicity values into a solar abundance scale. *Default*: Solar['total'] (see pyloser.py)
- The remaining **kwargs options are passed directly to fsps.StellarPopulations, so any stellar population available in python-FSPS can be generated. NOTE: sfh=0 and zcontinuous=1 should always be used.

If you have a lookup table and don't know the options used to generate it, you can list the `fsps_options` data block using the h5dump command at the system prompt:

```
% h5dump -d fsps_options my_new_SSP_table.hdf5
```

This will give you a bunch of hdf5 header info but at the end will be the `DATA` block which lists the FSPS options used.

1.7.6 Performance tips

- The code is cython parallelized over objects, so for efficiency it is best to run many objects within a single `photometry` instance. Try not to do a single galaxy at a time!
- Generally, computing the extinction and spectra takes most of the time; once the spectra are computed, applying bandpasses is fast. So it is also better to generate as many bands as possible in one call.

1.8 Aperture Quantities

`CAESAR` comes with a stand-alone function to sum quantities within a user-specified aperture around galaxies, called `get_aperture_masses()`. This operates directly on galaxies and halos from a `CAESAR` catalog, so must be used after the catalog has been generated.

`get_aperture_masses()` can compute masses for any particle type, or HI/H2/SFR. The aperture size can be specified as a constant for all galaxies, or an array of length the number of galaxies. This can be done in 3-D, or in 2-D projected along any principal axis. It is fully cython parallel.

Note that `get_aperture_masses()` only sums over particles within a galaxy's halo. For a large aperture, or for satellites near the edge of the halo, this may not give fully accurate answers. Also, this means 2-D projections are done through the entire halo.

1.8.1 Usage

This example computes the quantities listed in `myquants` in a 2-D aperture projected in `z`, within an aperture given by twice the stellar half mass radius for each galaxy, on 8 cores:

```
In [1]: import caesar
In [2]: from caesar.hydrogen_mass_calc import get_aperture_masses
In [3]: sim = caesar.load(CAESARFILE)
In [4]: myquants = ['gas','star','dm','sfr','HI','H2']
In [5]: rhalf = np.array([i.radii['stellar_half_mass'] for i in sim.galaxies])
In [6]: m_apert = get_aperture_masses(SNAPFILE,sim.galaxies,sim.halos,
↪quantities=myquants,aperture=2*rhalf,projection='z',nproc=8)
```

`get_aperture_masses()` returns a 2-D array of size `(Nquants,Ngal)`, with the aperture-summed quantities for each galaxy, in the order specified in the `quantities` option. `CAESARFILE` and `SNAPFILE` are the filenames of the `CAESAR` catalog and particle snapshot, respectively.

1.8.2 Options

`get_aperture_masses()` requires the original simulation snapshot, as well as the `galaxies` and `halos` objects from the corresponding `CAESAR` catalog. It operates only on galaxies, and cannot operate on a subset of objects.

The following options can be passed to `get_aperture_masses()`:

- `quantities`: Can be any particle type (e.g. `'gas'`), or else `'HI'`, `'H2'` or `'sfr'`. *Default:* `['gas','star', 'dm']`
- `aperture`: The aperture size. Can be a constant, which is assumed to be in comoving kpc, or else an array of length `Ngalaxies`. *Default:* `30`
- `projection`: `None` gives the 3-D aperture values. Specifying `'x'`, `'y'`, `'z'` gives the 2-D aperture projected along that axis. *Default:* `None`
- `exclude`: `None` includes all particles in halo. `satellites` or `central` excludes particles in satellite or central galaxies, respectively. `galaxies/both` excludes all particles in galaxies. *Default:* `None`
- `nproc`: Number of OpenMP cores (using joblib, passed as `n_jobs`). *Default:* 1

The routine returns a single 2-D array of length `(Nquants,Ngal)`, where `Nquants=len(quantities)` and `Ngal=len(sim.galaxies)`.

1.9 Units

`CAESAR` leverages yt's symbolic units. Every meaningful quantity *should* have a unit attached to it. One of this advantages this provides is that you no longer have to keep track of little *h* or remembering if you are dealing with comoving or physical coordinates.

1.9.1 Working with units

Let's take a look at some quick examples of what the units look like, and how we might take advantage of the easy conversion methods. Say we have a `CAESAR` object with `obj.simulation.redshift=2` . `Caesar` generally defaults its length units to be comoving kpc (`kpccm`):

```
In [1]: obj.galaxies[0].radii['total']
Out[1]: 22.2676089684 kpccm
```

Note the `cm` tacked on, which stands for *comoving*.

Because this particular galaxy is at *z=2* we may want to convert that radius to *physical* kiloparsecs:

```
In [2]: obj.galaxies[0].radii['total'].to('kpc')
Out[2]: 7.42253557308 kpc
```

or to physical *kpc/h* (using `obj.simulation.hubble_constant=0.7`):

```
In [3]: obj.galaxies[0].radii['total'].to('kpc/h')
Out[3]: 5.19577490116 kpc/h
```

When adding and subtracting quantities, they will be all be converted to the units of the first quantity. You don't have to worry about homogenizing the units yourself!

1.9.2 Assigning units

Quantities that are added or subtracted must have convertible units. This means you cannot add a simple number to a quantity with symbolic units; you must first assign a unit to that number (or array).

To assign a unit, you can use the yt functions `YTQuantity` and `YTArray`:

```
In [4]: from yt import YTQuantity
In [5]: x = YTQuantity(10, 'Mpc')
In [6]: print(x.to('kpc'))
Out[6]: 10000.0 kpc
```

Similarly, use `YTArray` for arrays.

1.9.3 Removing units

If you need to get rid of the units and return a value for any reason, simply append `.d` to the quantity:

```
In [7]: print(x.d)
Out[7]: 10
In [8]: print(x.to('kpc').d)
Out[8]: 10000.0
```

For further information and tutorials regarding yt's units please visit the symbolic unit page.

The various units and unit systems that are available in yt are described here .

Below you will find references for the various modules and functions contained within the `CAESAR` package. This information is all pulled from docstrings within the source code and can also be accessed from within python by putting a ? after the function.

1.10 Code Reference

1.10.1 CAESAR Module

class `main.`**CAESAR**(*ds=0*, **args*, ***kwargs*)

Bases: `object`

Master CAESAR class.

CAESAR objects contain all references to halos, galaxies, and clouds for a single snapshot. Its output format is portable and global object statistics can be examined without the raw simulation file.

Parameters

- **ds** (*yt dataset, optional*) – A dataset via `ds = yt.load(snapshot)`

- **mass** (*str, optional*) – Mass unit to store data with. Defaults to 'Msun'.
- **length** (*str, optional*) – Length unit to store data with. Defaults to 'kpccm'.
- **velocity** (*str, optional*) – Velocity unit to store data with. Defaults to 'km/s'.
- **time** (*str, optional*) – Time unit to store data with. Defaults to 'yr'.
- **temperature** (*str, optional*) – Temperature unit to store data with. Defaults to 'K'.

Examples

```
>>> import caesar
>>> obj = caesar.CAESAR()
```

cloudinfo(*top=10*)
Method to print general info for the most massive clouds identified via CAESAR.

Parameters **top** (*int, optional*) – Number of results to print. Defaults to 10.

Notes

This prints to terminal, and is meant for use in an interactive session.

property data_manager
On demand DataManager class.

galinfo(*top=10*)
Method to print general info for the most massive galaxies identified via CAESAR.

Parameters **top** (*int, optional*) – Number of results to print. Defaults to 10.

Notes

This prints to terminal, and is meant for use in an interactive session.

haloinfo(*top=10*)
Method to print general info for the most massive halos identified via CAESAR.

Parameters **top** (*int, optional*) – Number of results to print. Defaults to 10.

Notes

This prints to terminal, and is meant for use in an interactive session.

member_search(**args, **kwargs*)
Meat and potatoes of CAESAR.

This method is responsible for loading particle/field data from disk, creating halos, galaxies and clouds, linking objects together, and finally calculating HI/H2 masses if necessary.

Parameters

- **unbind_halos** (*boolean, optional*) – Unbind halos? Defaults to False
- **unbind_galaxies** (*boolean, optional*) – Unbind galaxies? Defaults to False
- **b_halo** (*float, optional*) – Quantity used in the linking length (LL) for halos. LL = mean_interparticle_separation * b_halo. Defaults to `b_halo = 0.2`.

- **b_galaxy** (*float, optional*) – Quantity used in the linking length (LL) for galaxies. LL = mean_interparticle_separation * b_galaxy. Defaults to `b_galaxy = b_halo * 0.2`.
- **ll_cloud** (*float, optional*) – Quantity used in the linking length (LL) for clouds in comoving kpc (kpccm).
- **fofclouds** (*boolean,optional*) – Indicates if we're running 3D fof on clouds. Default is that this is set to false
- **fof6d** (*boolean, optional*) – Indicates if we're running galaxy finding with 6D FOF vs the default of 3D FOF
- **fof6d_LL_factor** (*float, optional*) – Sets linking length for fof6d
- **fof6d_mingrp** (*float, optional*) – Sets minimum group size for fof6d
- **fof6d_velLL** (*float, optional*) – Sets linking length for velocity in fof6d
- **nproc** (*int, optional*) – Sets number of processors for fof6d and progen_rad
- **blackholes** (*boolean, optional*) – Indicate if blackholes are present in your simulation. This must be toggled on manually as there is no clear cut way to determine if PartType5 is a low-res particle, or a black hole.
- **dust** (*boolean, optional*) – Indicate if active dust particles are present in your simulation. This must be toggled on manually as there is no clear cut way to determine if PartType3 is a low-res particle, or an active dust particle.
- **lowres** (*list, optional*) – If you are running `CAESAR` on a Gadget/GIZMO zoom simulation in HDF5 format, you may want to check each halo for low-resolution contamination. By passing in a list of particle types (ex. [2,3,5]) we will check ALL objects for contamination and add the `contamination` attribute to all objects. Search distance defaults to 2.5x radii['total'].

Examples

```
>>> obj.member_search(blackholes=False)
```

reset_default_returns(*group_type='all'*)

Reset the default returns for object dictionaries.

This function resets the default return quantities for CAESAR halo/galaxy/cloud objects including `mass`, `radius`, `sigma`, `metallicity`, and `temperature`.

Parameters

- **obj** (*main.CAESAR*) – Main CAESAR object.
- **group_type** (*{'all', 'halo', 'galaxy', 'cloud'}, optional*) – Group to reset return values for.

save(*filename*)

Save CAESAR file.

Parameters **filename** (*str*) – The name of the output file.

Examples

```
>>> obj.save('output.hdf5')
```

`set_default_cloud_returns`(*category*, *value*)
Set the default return quantity for a given cloud attribute.

Parameters

- **category** (*str*) – The attribute to redirect to a different quantity.
- **value** (*str*) – The internal name of the new quantity which must be present in the dictionary

`set_default_galaxy_returns`(*category*, *value*)
Set the default return quantity for a given galaxy attribute.

Parameters

- **category** (*str*) – The attribute to redirect to a different quantity.
- **value** (*str*) – The internal name of the new quantity which must be present in the dictionary

`set_default_halo_returns`(*category*, *value*)
Set the default return quantity for a given halo attribute.

Parameters

- **category** (*str*) – The attribute to redirect to a different quantity.
- **value** (*str*) – The internal name of the new quantity which must be present in the dictionary

`vtk_vis`(***kwargs*)
Method to visualize an entire simulation with VTK.

Parameters

- **obj** (*main.CAESAR*) – Simulation object to visualize.
- **ptypes** (*list*) – List containing one or more of the following: 'dm','gas','star', which dictates which particles to render.
- **halo_only** (*boolean*) – If True only render particles belonging to halos.
- **galaxy_only** (*boolean*) – If True only render particles belonging to galaxies. Note that this overwrites `halo_only`.
- **annotate_halos** (*boolean, list, int, optional*) – Add labels to the render at the location of halos annotating the group ID and total mass. If True then all halos are annotated, if an integer list then halos of those indexes are annotated, and finally if an integer than the most massive N halos are annotated.
- **annotate_galaxies** (*boolean, list, int, optional*) – Add labels to the render at the location of galaxies annotating the group ID and total mass. If True then all galaxies are annotated, if an integer list then galaxies of those indexes are annotated, and finally if an integer than the most massive N galaxies are annotated.

`property yt_dataset`
The yt dataset to perform actions on.

1.10.2 FUBAR

1.10.3 Group Class

class group.**Cloud**(*obj*)

Bases: `group.Group`

Cloud class which has the central boolean.

class group.**Galaxy**(*obj*)

Bases: `group.Group`

Galaxy class which has the central boolean.

class group.**Group**(*obj*)

Bases: `object`

Parent class for halo and galaxy and halo objects.

_assign_local_data()

Assign glist/slist/dmlist/bhlist/dlist for this group. Also sets the ngas/nstar/ndm/nbh/ndust attributes.

_calculate_angular_quantities()

Calculate angular momentum, spin, max_vphi and max_vr.

_calculate_center_of_mass_quantities()

Calculate center-of-mass position and velocity. From caesar_mika

_calculate_gas_quantities()

Calculate gas quantities: SFR/Metallicity/Temperature.

_calculate_masses()

Calculate various total masses.

_calculate_radial_quantities()

Calculate various component radii and half radii

_calculate_star_quantities()

Calculate star quantities: Metallicity, …

_calculate_total_mass()

Calculate the total mass of the object.

_calculate_velocity_dispersions()

Calculate velocity dispersions for the various components.

_calculate_virial_quantities()

Calculates virial quantities such as r200, circular velocity, and virial temperature.

_cleanup()

cleanup function to delete attributes no longer needed

_delete_attribute(*a*)

Helper method to delete an attribute if present.

_delete_key(*d*, *k*)

Helper method to delete a dict key.

_process_group()

Process each group after creation. This entails calculating the total mass, iteratively unbinding (if enabled), then calculating more masses, radial quants, virial quants, velocity dispersions, angular quants, and final gas quants.

`_remove_dm_references()`
Galaxies/clouds do not have DM, so remove references.

`_unbind()`
Iterative procedure to unbind objects.

`property _valid`
Check against the minimum number of particles to see if this object is 'valid'.

`contamination_check`(*lowres=[2, 3, 5]*, *search_factor=1.0*, *printer=True*)
Check for low resolution particle contamination.

This method checks for low-resolution particles within `search_factor` of the maximum halo radius. When this method is called on a galaxy, it refers to the parent halo.

Parameters

- **lowres** (*list, optional*) – Particle types to be considered low-res. Defaults to [2,3,5]; if your simulation contains blackholes you will want to pass in [2,3]; if your simulation contains active

  dust particles you will not include 3.
- **search_factor** (*float, optional*) – Factor to expand the maximum halo radius search distance by. Default is 2.5
- **printer** (*boolean, optional*) – Print results?

Notes

This method currently ONLY works on GADGET/GIZMO HDF5 files.

`info()`
Method to quickly print out object attributes.

`vtk_vis`(*rotate=False*)
Method to render this group's points via VTK.

Parameters **rotate** (*boolean*) – Align angular momentum vector with the z-axis before rendering?

Notes

Opens up a pyVTK window; you must have VTK installed to use this method. It is easiest to install via `conda install vtk`.

`write_IC_mask`(*ic_ds*, *filename*, *search_factor=2.5*, *radius_type='total'*)
Write MUSIC initial condition mask to disk. If called on a galaxy it will look for the parent halo in the IC.

Parameters

- **ic_ds** (*yt dataset*) – The initial condition dataset via `yt.load()`.
- **filename** (*str*) – The filename of which to write the mask to. If a full path is not supplied then it will be written in the current directory.
- **search_factor** (*float, optional*) – How far from the center to select DM particles. Default is 2.5
- **print_extents** (*bool, optional*) – Print MUSIC extents for cuboid after mask creation

Examples

```
>>> import yt
>>> import caesar
>>>
>>> snap = 'my_snapshot.hdf5'
>>> ic   = 'IC.dat'
>>>
>>> ds    = yt.load(snap)
>>> ic_ds = yt.load(ic)
>>>
>>> obj = caesar.load('caesar_my_snapshot.hdf5', ds)
>>> obj.galaxies[0].write_IC_mask(ic_ds, 'mymask.txt')
```

class group.**GroupList**(*name*)

Bases: `object`

Class to hold particle/field index lists.

class group.**GroupProperty**(*source_dict*, *name*)

Bases: `object`

Class to return default values for the quantities held in the category_mapper dictionaries.

class group.**Halo**(*obj*)

Bases: *group.Group*

Halo class which has the dmlist attribute, and child boolean.

group.**create_new_group**(*obj*, *group_type*)

Simple function to create a new instance of a specified *group.Group*.

Parameters

- **obj** (*main.CAESAR*) – Main caesar object.
- **group_type** (*{'halo', 'galaxy','cloud'}*) – Which type of group? Options are: *halo* and *galaxy*.

Returns **group** – Subclass *group.Halo* or *group.Galaxy*.

Return type *group.Group*

1.10.4 Data Manager

class data_manager.**DataManager**(*obj*)

Bases: `object`

Class to handle the initial IO and data storage for the duration of a CAESAR run.

Parameters **obj** (*main.CAESAR*) – Main CAESAR object.

load_particle_data(*select=None*)

Loads positions, velocities, masses, particle types, and indexes. Assigns a global glist, slist, dlist, dmlist, and bhlist used throughout the group analysis. Finally assigns ngas/nstar/ndm/nbh values.

1.10.5 Property Getter

1.10.6 Assignment and Linking

Assignment

assignment.**assign_central_galaxies**(*obj*, *central_mass_definition='total'*)

Assign central galaxies.

Iterate through halos and consider the most massive galaxy within a central and all other satellites.

Parameters **obj** (*main.CAESAR*) – Object containing the galaxies to assign centrals. Halos must already be assigned via *assign_galaxies_to_halos*.

assignment.**assign_clouds_to_galaxies**(*obj*)

Assign clouds to galaxies.

This function compares cloud_glist with galaxy_glist to determine which galaxy the majority of particles within each cloud lies. Finally we assign the .clouds list to each galaxy.

Parameters **obj** (*main.CAESAR*) – Object containing the galaxies and halos lists.

assignment.**assign_galaxies_to_halos**(*obj*)

Assign galaxies to halos.

This function compares galaxy_glist + galaxy_slist with halo_glist + halo_slist to determine which halo the majority of particles within each galaxy lie. Finally we assign the .galaxies list to each halo.

Parameters **obj** (*main.CAESAR*) – Object containing the galaxies and halos lists.

Linking

linking.**create_sublists**(*obj*)

Create sublists of objects.

Will create the sublists:

- central_galaxies
- satellite_galaxies
- unassigned_galaxies (those without a halo)

Parameters **obj** (*main.CAESAR*) – Object containing halos and galaxies lists already linked.

linking.**link_clouds_and_galaxies**(*obj*)

Link clouds and galaxies to one another.

This function creates the links between cloud–>galaxy and galaxy–>cloud objects. Is run during creation, and loading in of each CAESAR file.

Parameters **obj** (*main.CAESAR*) – Object containing halos and galaxies lists.

linking.**link_galaxies_and_halos**(*obj*)

Link galaxies and halos to one another.

This function creates the links between galaxy–>halo and halo–>galaxy objects. Is run during creation, and loading in of each CAESAR file.

Parameters **obj** (*main.CAESAR*) – Object containing halos and galaxies lists.

1.10.7 Misc. Utilities

utils.**calculate_local_densities**(*obj*, *group_list*)

Calculate the local number and mass density of objects.

Parameters

- **obj** (*SPHGR object*) –
- **group_list** (*list*) – List of objects to perform this operation on.

utils.**info_printer**(*obj*, *group_type*, *top*)

General method to print data.

Parameters

- **obj** (*main.CAESAR*) – Main CAESAR object.
- **group_type** (*{'halo', 'galaxy', 'cloud'}*) – Type of group to print data for.
- **top** (*int*) – Number of objects to print.

utils.**memlog**(*msg*)

utils.**rotator**(*vals*, *ALPHA=0*, *BETA=0*)

Rotate particle set around given angles.

Parameters

- **vals** (*np.array*) – a Nx3 array typically consisting of either positions or velocities.
- **ALPHA** (*float, optional*) – First angle to rotate about
- **BETA** (*float, optional*) – Second angle to rotate about

Examples

```
>>> rotated_pos = rotator(positions, 32.3, 55.2)
```

1.10.8 External Group Functions

group_funcs.**get_full_mass_radius**(*radii*, *ptype*, *binary*)

Get full mass radius for a set of particles.

Parameters

- **radii** (*np.ndarray[::-1]*) – Radii of particles
- **ptype** (*np.ndarray[::-1]*) – Array of integers containing the particle types.
- **binary** (*int*) – Integer used to select particle types. For example, if you are interested in particle types 0 and 3 this value would be 2^0+2^3=9.

Notes

This function iterates forward through the array, so it is advisable to reverse the radii & ptype arrays before passing them via np.ndarray[::-1].

`group_funcs.`**`get_half_mass_radius`**`(`*mass*, *radii*, *ptype*, *half_mass*, *binary*`)`

Get half mass radius for a set of particles.

Parameters

- **mass** (*np.ndarray*) – Masses of particles.
- **radii** (*np.ndarray*) – Radii of particles.
- **ptype** (*np.ndarray*) – Array of integers containing the particle types.
- **half_mass** (*double*) – Half mass value to accumulate to.
- **binary** (*int*) – Integer used to select particle types. For example, if you are interested in particle types 0 and 3 this value would be 2^0+2^3=9.

`group_funcs.`**`get_periodic_r`**`(`*boxsize*, *center*, *pos*, *r*`)`

Get periodic radii.

Parameters

- **boxsize** (*double*) – The size of your domain.
- **center** (*np.ndarray([x,y,z])*) – Position in which to calculate the radius from.
- **pos** (*np.ndarray*) – Nx3 numpy array containing the positions of particles.
- **r** (*np.array*) – Empty array to fill with radius values.

`group_funcs.`**`get_virial_mr`**`(`*Density*, *r*, *mass*, *collectRadii*`)`

Get virial mass and radius.

Parameters

- **Density** (*array*) – Different densities you are interested in: e.g rho200, rhovirial, … They have to be in ascending order.
- **r** (*array*) – Particle radii inward
- **mass** (*array*) – Cumulative Particle masses inward
- **collectRadii** (*array*) – Empty array to contain the radii Should be the same size as the Densities

`group_funcs.`**`rotator`**`(`*vals*, *Rx*, *Ry*, *ALPHA*, *BETA*`)`

Rotate a number of vectors around ALPHA, BETA

Parameters

- **vals** (*np.ndarray*) – Nx3 np.ndarray of values you want to rotate.
- **Rx** (*np.ndarray*) – 3x3 array used for the first rotation about ALPHA. The dot product is taken against each value: vals[i] = np.dot(Rx, vals[i])
- **Ry** (*np.ndarray*) – 3x3 array used for the second rotation about BETA The dot product is taken against each value: vals[i] = np.dot(Ry, vals[i])
- **ALPHA** (*double*) – Angle to rotate around first.
- **BETA** (*double*) – Angle to rotate around second.

Notes

This is typically called from `utils.rotator()`.

1.10.9 HI/H2 Mass Calc

hydrogen_mass_calc.**assign_halo_gas_to_galaxies**(*internal_galaxy_pos*, *internal_galaxy_mass*, *internal_glist*, *internal_galaxy_index_list*, *galaxy_glist*, *grhoH*, *gpos*, *galaxy_HImass*, *galaxy_H2mass*, *HImass*, *H2mass*, *low_rho_thresh*, *boxsize*, *halfbox*)

Function to assign halo gas to galaxies.

When we assign galaxies in CAESAR, we only consider dense gas. But when considering HI gas however, it is often desirable to also consider low-density gas 'outside' of the galaxy. This function calculates the mass weighted distance to each galaxy within a given halo and assigns low-density gas to the 'nearest' galaxy.

Typically called from `hydrogen_mass_calc.hydrogen_mass_calc()`.

hydrogen_mass_calc.**check_values**(*obj*)

Check to make sure that we have the required fields available to perform the hydrogen mass frac calculation.

Parameters **obj** (`main.CAESAR`) – Main CAESAR object.

Returns Returns True if all fields are present, False otherwise.

Return type bool

hydrogen_mass_calc.**hydrogen_mass_calc**(*obj*, ***kwargs*)

Calculate the neutral and molecular mass contents of SPH particles.

For non star forming gas particles assigned to halos we calculate the neutral fraction based on equations from Popping+09 and Rahmati+13. If H2 block is not present in the simulation file we estimate the neutral and molecular fraciton via Leroy+08. Once these fractions are calculated we assign HI/H2 masses to galaxies & halos based on their mass-weighted distances.

Parameters **obj** (`main.CAESAR`) – Main CAESAR object.

Returns **HImass, H2mass** – Contains the HImass and H2 mass of each individual particle.

Return type np.ndarray, np.ndarray

1.10.10 Saving and Loading

Saver

saver.**check_and_write_dataset**(*obj*, *key*, *hd*)

General function for writing an HDF5 dataset.

Parameters

- **obj** (`main.CAESAR`) – Main caesar object to save.
- **key** (*str*) – Name of dataset to write.
- **hd** (*h5py.Group*) – Open HDF5 group.

saver.**save**(*obj*, *filename='test.hdf5'*)

Function to save a CAESAR file to disk.

Parameters

- **obj** (*main.CAESAR*) – Main caesar object to save.
- **filename** (*str, optional*) – Filename of the output file.

Examples

```
>>> obj.save('output.hdf5')
```

saver.**serialize_attributes**(*obj_list*, *hd*, *hd_dicts*)
Function that goes through a list full of halos/galaxies/clouds and serializes their attributes.

Parameters

- **obj** (*main.CAESAR*) – Main caesar object.
- **hd** (*h5py.Group*) – Open HDF5 group for lists.
- **hd_dicts** (*h5py.Group*) – Open HDF5 group for dictionaries.

saver.**serialize_global_attribs**(*obj*, *hd*)
Function that goes through a caesar object and saves general attributes.

Parameters

- **obj** (*main.CAESAR*) – Main caesar object.
- **hd** (*h5py.File*) – Open HDF5 dataset.

saver.**serialize_list**(*obj_list*, *key*, *hd*)
Function that serializes a index list (glist/etc) for objects.

Parameters

- **obj** (*main.CAESAR*) – Main caesar object.
- **key** (*str*) – Name of the index list.
- **hd** (*h5py.Group*) – Open HDF5 group.

Loader

This module is a lazy replacement for caesar.loader

Instead of eagerly constructing every Halo, Galaxy, and Cloud, this module provides a class which lazily constructs Groups and their attributes only as they are accessed, and caches them using functools.lru_cache.

The design of this module was motivated by profiling the previous eager loader, which revealed these things dominated load time, in order of importance: 1) Creating unyt.unty_quantity objects 2) Creating Halo/Galaxy/Cloud objects 3) Reading datasets from the HDF5 file Therefore, this module avoids creating quantities as much as possible and caches them. It might be nice to only load part of the backing HDF5 datasets, but that stage is already quite fast and it looks to me like the HDF5 library (or at least h5py) has some minimum granularity at which it will pull data off disk which is ~1M items, which at the time of writing (April 21, 2020) exceeds the size of most datasets in caesar files, including from the m100n1024 SIMBA run I've been testing with.

class loader.**CAESAR**(*filename*, *skip_hash_check=False*)
Bases: `object`

property central_galaxies

cloudinfo(*top=10*)

galinfo(*top=10*)

haloinfo(*top=10*)

property satellite_galaxies

property yt_dataset
The yt dataset to perform actions on.

class loader.**Cloud**(*obj*, *index*)
Bases: `loader.Group`

property dlist

property glist

class loader.**Galaxy**(*obj*, *index*)
Bases: `loader.Group`

property bhlist

property cloud_index_list

property clouds

property dlist

property glist

property satellites

property slist

class loader.**Group**
Bases: `object`

contamination_check(*lowres=[2, 3, 5]*, *search_factor=2.5*, *printer=True*)

info()

property mass

property metallicity

property temperature

write_IC_mask(*ic_ds*, *filename*, *search_factor=2.5*, *radius_type='total'*)

class loader.**Halo**(*obj*, *index*)
Bases: `loader.Group`

property bhlist

property central_galaxy

property dlist

property dm2list

property dm3list

property dmlist

property galaxies

property galaxy_index_list

property glist

property satellite_galaxies

property slist

class loader.**LazyDataset**(*obj*, *dataset_path*)

Bases: `object`

A lazily-loaded HDF5 dataset

class loader.**LazyDict**(*keys*, *builder*)

Bases: `collections.abc.Mapping`

This type should be indistinguishable from the built-in dict. Any observable difference except the explicit type and performance is considered a bug.

The implementation wraps a dict which initially maps every key to None, and are replaced by calling the passed-in callable as they are accessed.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

values() → an object providing a view on D's values

class loader.**LazyList**(*length*, *builder*)

Bases: `collections.abc.Sequence`

This type should be indistinguishable from the built-in list. Any observable difference except the explicit type and performance is considered a bug.

The implementation wraps a list which is intially filled with None, which is very fast to create at any size because None is a singleton. The initial elements are replaced by calling the passed-in callable as they are accessed.

count(*value*) → integer -- return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

loader.**load**(*filename*, *skip_hash_check=False*)

1.10.11 Driver

class driver.**Snapshot**(*snapdir*, *snapname*, *snapnum*, *extension*)

Bases: `object`

Class for tracking paths and data for simulation snapshots.

Parameters

- **snapdir** (*str*) – Path to snapshot
- **snapname** (*str*) – Name of snapshot minus number and extension
- **snapnum** (*int*) – Snapshot number
- **extension** (*str, optional*) – File extension of your snapshot, 'hdf5' by default.

Notes

This class attempts to concat strings to form a full path to your simulation snapshot in the following manner:

```
>>> '%s/%s%03d.%s' % (snapdir, snapname, snapnum, extension)
```

_make_output_dir()
If output directory is not present, create it.

member_search(*skipran*, ***kwargs*)
Perform the member_search() method on this snapshot.

set_output_information(*ds*, *prefix='caesar_'*, *suffix='hdf5'*)
Set the name of the CAESAR output file.

driver.**drive**(*snapdirs*, *snapname*, *snapnums*, *progen=False*, *skipran=False*, *member_search=True*, *extension='hdf5'*, *caesar_prefix='caesar_'*, ***kwargs*)
Driver function for running `CAESAR` on multiple snapshots.

Can utilize mpi4py to run analysis in parallel given that `MPI` and `mpi4py` is correctly installed. To do this you must create a script similar to the example below, then execute it via:

```
>>> mpirun -np 8 python my_script.py
```

Parameters

- **snapdirs** (*str or list*) – A path to your snapshot directory, or a list of paths to your snapshot directories.
- **snapname** (*str*) – Formatting of your snapshot name disregarding any integer numbers or file extensions; for example: `snap_N256L16_`
- **snapnums** (*int or list or array*) – A single integer, a list of integers, or an array of integers. These are the snapshot numbers you would like to run CAESAR on.
- **progen** (*boolean, optional*) – Perform most massive progenitor search. Defaults to False.
- **skipran** (*boolean, optional*) – Skip running member_search() if CAESAR outputs are already present. Defaults to False.
- **member_search** (*boolean, optional*) – Perform the member_search() method on each snapshot. Defaults to True. This is useful to set to False if you want to just perform progen for instance.
- **extension** (*str, optional*) – Specify your snapshot file extension. Defaults to *hdf5*
- **prefix** (*str, optional*) – Specify prefix for caesar filename (replaces '**snap_**')
- **unbind_halos** (*boolean, optional*) – Unbind halos? Defaults to False
- **unbind_galaxies** (*boolean, optional*) – Unbind galaxies? Defaults to False
- **b_halo** (*float, optional*) – Quantity used in the linking length (LL) for halos. LL = mean_interparticle_separation * b_halo. Defaults to `b_halo = 0.2`.
- **b_galaxy** (*float, optional*) – Quantity used in the linking length (LL) for galaxies. LL = mean_interparticle_separation * b_galaxy. Defaults to `b_galaxy = b_halo * 0.2`.
- **ll_cloud** (*float, optional*) – Linking length in comoving kpc (**kpccm_** for clouds. Defaults to same linking length as used for galaxies.

- **fofclouds** (*boolean, optional*) – Sets whether or not we run 3D FOF for clouds. Default is that this is not run as this isn't the typical use case for Caesar, and slows things down a bit
- **fof6d** (*boolean, optional*) – Sets whether or not we do 6D FOF for galaxies. if not set, the default is to do normal 3D FOF for galaxies.
- **fof6d_LL_factor** (*float, optional*) – Sets linking length for fof6d
- **fof6d_mingrp** (*float, optional*) – Sets minimum group size for fof6d
- **fof6d_velLL** (*float, optional*) – Sets linking length for velocity in fof6d
- **nproc** (*int, optional*) – Sets number of processors for fof6d
- **blackholes** (*boolean, optional*) – Indicate if blackholes are present in your simulation. This must be toggled on manually as there is no clear cut way to determine if PartType5 is a low-res particle, or a black hole.
- **lowres** (*list, optional*) – If you are running `CAESAR` on a Gadget/GIZMO zoom simulation in HDF5 format, you may want to check each halo for low-resolution contamination. By passing in a list of particle types (ex. [2,3,5]) we will check ALL objects for contamination and add the `contamination` attribute to all objects. Search distance defaults to 2.5x radii['total'].

Examples

```
>>> import numpy as np
>>> snapdir  = '/Users/bob/Research/N256L16/some_sim'
>>> snapname = 'snap_N256L16_'
>>> snapnums = np.arange(0,86)
>>>
>>> import caesar
>>> caesar.drive(snapdir, snapname, snapnums, skipran=False, progen=True)
```

`driver.`**`print_art`**`()`
Print some ascii art.

1.10.12 Progen

`progen.`**`caesar_filename`**`(`*snap*, *prefix*, *extension*`)`
return full Caesar filename including filetype extension for given Snapshot object.

`progen.`**`check_if_progen_is_present`**`(`*caesar_file*, *index_name*`)`
Check CAESAR file for progen indexes.

Parameters

- **caesar_file** (*str*) – Name (including path) of Caesar file with tree_data
- **index_name** (*str*) – Name of progen index to get redshift for (e.g. 'progen_galaxy_star')

`progen.`**`collect_group_IDs`**`(`*obj*, *data_type*, *part_type*, *snap_dir*`)`
Collates list of particle and associated group IDs for all specified objects. Returns particle and group ID lists, and a hash list of indexes for particle IDs corresponding to the starting index of each group.

Parameters

- **obj** (*main.CAESAR*) – Caesar object for which to collect group IDs

- **data_type** (*str*) – 'halo', 'galaxy', or 'cloud'
- **part_type** (*str*) – Particle type
- **snap_dir** (*str*) – Path where snapshot files are located; if None, uses obj.simulation.fullpath

progen.**find_progens**(*pid_current*, *pid_target*, *gid_current*, *gid_target*, *pid_hash*, *npart_target*, *n_most=None*, *min_in_common=0.1*, *nproc=1*, *reverse_match=False*)

Find most massive and second most massive progenitor/descendants.

Parameters

- **pids_current** (*np.ndarray*) – particle IDs from the current snapshot.
- **pids_target** (*np.ndarray*) – particle IDs from the previous/next snapshot.
- **gids_current** (*np.ndarray*) – group IDs from the current snapshot.
- **gids_target** (*np.ndarray*) – group IDs from the previous/next snapshot.
- **pid_hash** (*np.ndarray*) – indexes for the start of each group in pids_current
- **n_most** (*int*) – Find n_most most massive progenitors/descendants, None for all.
- **min_in_common** (*float*) – Require >this fraction of parts in common between object and progenitor to be a valid progenitor.
- **nproc** (*int*) – Number of cores for multiprocessing. Note that this doesn't help much since most of the time is spent in sorting.
- **reverse_match** (*bool*) –

progen.**get_progen_redshift**(*caesar_file*, *index_name*)

Returns redshift of progenitors/descendants currently stored in tree_data. Returns -1 (with warning) if no tree_data is found.

Parameters

- **caesar_file** (*str*) – Name (including path) of Caesar file with tree_data
- **index_name** (*str*) – Name of progen index to get redshift for (e.g. 'progen_galaxy_star')

progen.**progen_finder**(*obj_current*, *obj_target*, *caesar_file*, *snap_dir=None*, *data_type='galaxy'*, *part_type='star'*, *recompute=True*, *save=True*, *n_most=None*, *min_in_common=0.1*, *nproc=1*, *match_frac=False*, *reverse_match=False*)

Function to find the most massive progenitor of each Caesar object in obj_current in the previous snapshot. Returns list of progenitors in obj_target associated with objects in obj_current

Parameters

- **obj_current** (*main.CAESAR*) – Will search for the progenitors of the objects in this object.
- **obj_target** (*main.CAESAR*) – Looking for progenitors in this object.
- **caesar_file** (*str*) – Name (including path) of Caesar file associated with primary snapshot, where progen info will be written
- **snap_dir** (*str*) – Path where snapshot files are located; if None, uses obj.simulation.fullpath
- **data_type** (*str*) – 'halo', 'galaxy', or 'cloud'
- **part_type** (*str*) – Particle type in ptype_ints. Current options: 'gas', 'dm', 'dm2', 'star', 'bh'

- **recompute** (*bool*) – False = see if progen info exists in caesar_file and return, if not then compute True = always (re)compute progens
- **save** (*bool*) – True/False = write/do not write info to caesar_file
- **n_most** (*int*) – Find n_most most massive progenitors/descendants. Stored as a list for each galaxy. Default: None for all progenitors/descendants
- **min_in_common** (*float*) – Require >this fraction of parts in common between object and progenitor to be a valid progenitor.
- **nproc** (*int*) – Number of cores for multiprocessing.
- **match_fracs** (*bool*) – True/False = Return / do _not_ return match fraction for each match
- **reverse_match** (*bool*) – False = match all objects where fraction of _current_ is above min_in_common True = match all objects where fraction of _target_ is above min_in_common if match_fracs=True, returned fraction is the fraction of the current/target (False/True)

progen.**run_progen**(*snapdirs*, *snapname*, *snapnums*, *prefix='caesar_'*, *suffix='hdf5'*, ***kwargs*)

Function to run progenitor/descendant finder in specified snapshots (or redshifts) in a given directory.

Parameters

- **snapdirs** (*str or list of str*) – Full path of directory(s) where snapshots are located
- **snapname** (*str*) – Formatting of snapshot name excluding any integer numbers or file extensions; e.g. '**snap_N256L16_**'
- **snapnums** (*int or list of int*) – Snapshot numbers over which to run progen. Increasing order -> descendants; Decreasing -> progens.
- **prefix** (*str*) – Prefix for caesar filename; assumes these are in 'Groups' subdir
- **suffix** (*str*) – Filetype suffix for caesar filename
- **kwargs** (*Passed to progen_finder()*) –

progen.**wipe_progen_info**(*caesar_file*, *index_name=None*)

Remove all progenitor/descendant info from Caesar file.

Parameters

- **caesar_file** (*str*) – Name (including path) of Caesar file with tree_data
- **index_name** (*str (optional)*) – Name (or substring) of progen index to remove (e.g. 'progen_galaxy_star'). If not provided, removes *all* progen/descend info

progen.**z_to_snap**(*redshift*, *snaplist_file='Simba'*, *mode='closest'*)

Finds snapshot number and snapshot redshift close to input redshift.

Parameters

- **redshift** (*float*) – Redshift you want to find snapshot for
- **snaplist_file** (*str*) – Name (including path) of Caesar file with a list of expansion factors (in ascending order) at which snapshots are output. This is the same file as used when running a Gizmo/Gadget simulation. 'Simba' returns the value for the default Simba simulation snapshot list.
- **mode** (*str*) – 'closest' finds closest one in redshift 'higher'/'upper'/'above' finds the closest output >= redshift 'lower'/'below' finds the closest output <= redshift.

1.10.13 VTK Visualization

There are some built in methods to visualize particle clouds via VTK. These require the python-vtk wrapper to be installed. Unfortunately, compiling this wrapper manually is quite painful - I highly suggest you utilize the `conda` package manager to take care of this one for you via:

```
$> conda install vtk
```

Afterwards the VTK methods described below should work.

VTK Functions

These are the exposed VTK methods for both the *main.CAESAR* and *group.Group* objects.

`vtk_funcs.`**`group_vis`**(*group*, *rotate=True*)

Function to visualize a *group.Group* with VTK.

Parameters

- **group** (*group.Group*) – Group to visualize.
- **rotate** (*boolean*) – If true the positions are rotated so that the angular momentum vector is aligned with the z-axis.

`vtk_funcs.`**`sim_vis`**(*obj*, *ptypes=['dm', 'star', 'gas']*, *halo_only=True*, *galaxy_only=False*, *annotate_halos=False*, *annotate_galaxies=False*, *draw_spheres=None*)

Function to visualize an entire simulation with VTK.

Parameters

- **obj** (*main.CAESAR*) – Simulation object to visualize.
- **ptypes** (*list*) – List containing one or more of the following: 'dm','gas','star', which dictates which particles to render.
- **halo_only** (*boolean*) – If True only render particles belonging to halos.
- **galaxy_only** (*boolean*) – If True only render particles belonging to galaxies. Note that this overwrites `halo_only`.
- **annotate_halos** (*boolean, list, int, optional*) – Add labels to the render at the location of halos annotating the group ID and total mass. If True then all halos are annotated, if an integer list then halos of those indexes are annotated, and finally if an integer than the most massive N halos are annotated.
- **annotate_galaxies** (*boolean, list, int, optional*) – Add labels to the render at the location of galaxies annotating the group ID and total mass. If True then all galaxies are annotated, if an integer list then galaxies of those indexes are annotated, and finally if an integer than the most massive N galaxies are annotated.
- **draw_spheres** (*string, boolean*) – Add spheres around your annotated objects. The size is determined by the string you pass, should be from the .radii dict. If a boolean of True is passed it will use the total radii.

pyVTK Wrapper

This is a wrapper for python-vtk. You can utilize these methods to render other point data as well.

---

class vtk_vis.**vtk_render**

Bases: `object`

Base class for the vtk wrapper.

Keypress(*obj*, *event*)

draw_arrow(*p1*, *p2*, *shaft_r=0.01*, *tip_r=0.05*, *tip_l=0.2*, *balls=0*, *ball_color=[1, 1, 1]*, *ball_r=1*, *color=[1, 1, 1]*)

draw_cube(*center*, *size*, *color=[1, 1, 1]*)

Draw a cube in the scene.

Parameters

- **center** (*list or np.ndarray*) – Center of the box in 3D space.
- **size** (*float*) – How large the box should be on a side.
- **color** (*list, optional*) – Color of the outline in RGB.

draw_sphere(*pos*, *r*, *color=[1, 1, 1]*, *opacity=1*, *res=12*)

Draw a sphere in the scene.

Parameters

- **center** (*list or np.ndarray*) – Center of the sphere in 3D space.
- **r** (*float*) – Radius of the sphere.
- **color** (*list, optional*) – Color of the sphere in RGB.
- **opacity** (*float, optional*) – Transparency of the sphere.
- **res** (*int, optional*) – Resolution of the sphere.

makebutton()

place_label(*pos*, *text*, *text_color=[1, 1, 1]*, *text_font_size=12*, *label_box_color=[0, 0, 0]*, *label_box=1*, *label_box_opacity=0.8*)

Place a label in the scene.

Parameters

- **pos** (*tuple or np.ndarray*) – Position in 3D space where to place the label.
- **text** (*str*) – Label text.
- **text_color** (*list or np.ndarray, optional*) – Color of the label text in RGB.
- **text_font_size** (*int, optional*) – Text size of the label.
- **label_box_color** (*list or np.ndarray, optional*) – Background color of the label box in RGB.
- **label_box** (*int, optional*) – 0=do not show the label box, 1=show the label box.
- **label_box_opacity** (*float, optional*) – Opacity value of the background box (1=no transparency).

point_render(*pos*, *color=[1, 1, 1]*, *opacity=1*, *alpha=1*, *psize=1*)

Render a pointcloud in the scene.

Parameters

- **pos** (*np.ndarray*) – 3D positions of points.
- **color** (*list or np.ndarray, optional*) – Color of points. This can be a single RGB value or a list of RGB values (one per point).
- **opacity** (*float, optional*) – Transparency of the points.
- **alpha** (*float, optional*) – Transparency of the points (same as opacity).
- **psize** (*int, optional*) – Size of the points.

`quit()`

`render(xsize=800, ysize=800, bg_color=[0.5, 0.5, 0.5], focal_point=None, orient_widget=1)`

Final call to render the window.

Parameters

- **xsize** (*int, optional*) – Horizontal size of the window in pixels.
- **ysize** (*int, optional*) – Vertical size of the window in pixels.
- **bg_color** (*tuple or np.array, optional*) – Background color in RGB.
- **focal_point** (*tuple or np.array, optional*) – Where to focus the camera on rendering.
- **orient_widget** (*int, optional*) – Show the orient widget?

- genindex
- modindex
- search

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

S

T

U

V

W

Y

Z