yt_astro_analysis Documentation

Release 1.1.1

The yt project

Jan 28, 2022

CONTENTS:

This is `yt_astro_analysis`, the yt extension package for astrophysical analysis. This is primarily machinery that used to be in yt's analysis_modules. These were made into a separate package to allow yt to become less astro-specifc and to allow these modules to be developed on their own schedule.

CHAPTER

ONE

INSTALLATION

The most straightforward way to install `yt_astro_analysis` is to first install yt. This will take care of all `yt_astro_analysis` dependencies. After that, `yt_astro_analysis` can be installed with pip:

```
$ pip install yt_astro_analysis
```

If you use `conda` to manage packages, you can install `yt_astro_analysis` from conda-forge:

```
$ conda install -c conda-forge yt_astro_analysis
```

1.1 Installing from source

To install from source, it is still recommended to first install `yt` in the manner described above. Then, clone the git repository and install like this:

```
$ git clone https://github.com/yt-project/yt_astro_analysis
$ cd yt_astro_analysis
$ pip install -e .
```

1.2 Installing with Rockstar support

Note: As of `yt_astro_analysis` version 1.1, `yt_astro_analysis` runs with the most recent version of `rockstar-galaxies`. Older versions of `rockstar` will not work.

Rockstar support requires `yt_astro_analysis` to be installed from source. Before that, the `rockstar-galaxies` code must also be installed from source and the installation path then provided to `yt_astro_analysis`. Two recommended repositories exist for installing `rockstar-galaxies`, this one, by the original author, Peter Behroozi, and this one, maintained by John Wise.

Warning: If using Peter Behroozi's repository, the following command must be issued after loading the resulting halo catalog in `yt`:

```
ds = yt.load(...)
ds.parameters["format_revision"] = 2
```

To install `rockstar-galaxies`, do the following:

```
$ git clone https://bitbucket.org/jwise77/rockstar-galaxies
$ cd rockstar-galaxies
$ make lib
```

Then, go into the `yt_astro_analysis` source directory and add a file called "rockstar.cfg" with the path the `rockstar-galaxies` repo you just cloned. Then, install `yt_astro_analysis`.

```
$ cd yt_astro_analysis
$ echo <path_to_rockstar> > rockstar.cfg
$ pip install -e .
```

Finally, you'll need to make sure that the location of `librockstar-galaxies.so` is in your LD_LIBRARY_PATH.

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<path_to_rockstar>
```

CHAPTER

TWO

IMPORTING FROM YT_ASTRO_ANALYSIS

For every module that was moved from yt's analysis_modules to yt_astro_analysis, all imports can be changed simply by substituting `yt.analysis_modules` with `yt.extensions.astro_analysis`. For example, the following

```
from yt.analysis_modules.ppv_cube.api import PPVCube
```

becomes

```
from yt.extensions.astro_analysis.ppv_cube.api import PPVCube
```

CHAPTER

THREE

AVAILABLE MODULES

These are all the modules available in `yt_astro_analysis`.

3.1 Halo Analysis

This section covers finding and analyzing halos using the *`HaloCatalog`*. If you already have halo catalogs and simply want to load them into yt, see Halo Catalog Data.

3.1.1 Halo Finding

Halo finding and analysis are combined into a single framework called the *`HaloCatalog`*.

If you already have a halo catalog, either produced by one of the methods below or in a format described in Halo Catalog Data, and want to perform further analysis, skip to *Halo Analysis*.

Three halo finding methods exist within yt. These are:

- *FoF*: a basic friend-of-friends algorithm (e.g. Efstathiou et al. 1985)
- *HOP*: Eisenstein and Hut (1998).
- *Rockstar-galaxies*: a 6D phase-space halo finder that scales well, does substructure finding, and will automatically calculate halo ancestor/descendent links for merger trees (Behroozi et al. 2011).

Halo finding is performed through the creation of a *`HaloCatalog`* object. The dataset or datasets on which halo finding is to be performed should be loaded and given to the *`HaloCatalog`* along with the `finder_method` keyword to specify the method to be used.

```
import yt
from yt.extensions.astro_analysis.halo_analysis import HaloCatalog

data_ds = yt.load("Enzo_64/RD0006/RedshiftOutput0006")
hc = HaloCatalog(data_ds=data_ds, finder_method="hop")
hc.create()
```

Halo Finding on Multiple Snapshots

To run halo finding on a series of snapshots, provide a `DatasetSeries` or `SimulationTimeSeries` to the *`HaloCatalog`*. See Time Series Analysis and Analyzing an Entire Simulation for more information on creating these. All three halo finders can be run this way. If you want to make merger trees with Rockstar halo catalogs, you must run Rockstar in this way.

```
import yt
from yt.extensions.astro_analysis.halo_analysis import HaloCatalog

my_sim = yt.load_simulation("enzo_tiny_cosmology/32Mpc_32.enzo", "Enzo")
my_sim.get_time_series()
hc = HaloCatalog(data_ds=my_sim, finder_method="hop")
hc.create()
```

Halo Finder Options

The available `finder_method` options are "fof", "hop", or "rockstar". Each of these methods has their own set of keyword arguments to control functionality. These can specified in the form of a dictionary using the `finder_kwargs` keyword.

```
import yt
from yt.extensions.astro_analysis.halo_analysis import HaloCatalog

data_ds = yt.load("Enzo_64/RD0006/RedshiftOutput0006")
hc = HaloCatalog(
    data_ds=data_ds,
    finder_method="fof",
    finder_kwargs={"ptype": "stars", "padding": 0.02},
)
hc.create()
```

For a full list of options for each halo finder, see:

- FoF ("fof"): *`FOFHaloFinder`*
- HOP ("hop"): *`HOPHaloFinder`*
- Rockstar-galaxies ("rockstar"): *`RockstarHaloFinder`*

FoF

This is a basic friends-of-friends algorithm. Any two particles separated by less than a linking length are considered to be in the same group. See Efstathiou et al. (1985) for more details as well as `FOFHaloFinder`.

HOP

This is the method introduced by Eisenstein and Hut (1998). The procedure is roughly as follows.

1. Estimate the local density at each particle using a smoothing kernel.
2. Build chains of linked particles by 'hopping' from one particle to its densest neighbor. A particle which is its own densest neighbor is the end of the chain.
3. All chains that share the same densest particle are grouped together.
4. Groups are included, linked together, or discarded depending on the user-supplied over density threshold parameter. The default is 160.

For both the FoF and HOP halo finders, the resulting halo catalogs will be written to a directory associated with the `output_dir` keyword provided to the *`HaloCatalog`*. The number of files for each catalog is equal to the number of processors used. The catalog files have the naming convention `<dataset_name>/<dataset_name>.<processor_number>.h5`, where `dataset_name` refers to the name of the snapshot. For more information on loading these with yt, see YTHaloCatalog.

Rockstar-galaxies

Rockstar uses an adaptive hierarchical refinement of friends-of-friends groups in six phase-space dimensions and one time dimension, which allows for robust (grid-independent, shape-independent, and noise- resilient) tracking of substructure. The methods are described in Behroozi et al. 2011.

The `yt_astro_analysis` package works with the latest version of `rockstar-galaxies`. See *Installing with Rockstar support* for information on obtaining and installing `rockstar-galaxies` for use with `yt_astro_analysis`.

To run Rockstar, your script must be run with `mpirun` using a minimum of three processors. Rockstar processes are divided into three groups:

- readers: these read particle data from the snapshots. Set the number of readers with the `num_readers` keyword argument.
- writers: these perform the halo finding and write the subsequent halo catalogs. Set the number of writers with the `num_writers` keyword argument.
- server: this process coordinates the activity of the readers and writers. There is only one server process. The total number of processes given with `mpirun` must be equal to the number of readers plus writers plus one (for the server).

```
import yt

yt.enable_parallelism()
from yt.extensions.astro_analysis.halo_analysis import HaloCatalog

my_sim = yt.load_simulation("enzo_tiny_cosmology/32Mpc_32.enzo", "Enzo")
my_sim.get_time_series()
hc = HaloCatalog(
    data_ds=my_sim,
    finder_method="rockstar",
    finder_kwargs={"num_readers": 1, "num_writers": 1},
)
hc.create()
```

Warning: Running Rockstar from yt on multiple compute nodes connected by an Infiniband network can be problematic. It is recommended to force the use of the non-Infiniband network (e.g. Ethernet) using this flag: `--mca btl ^openib`. For example, to run with 24 cores, do: `mpirun -n 24 --mca btl ^openib python ./run_rockstar.py`.

See *RockstarHaloFinder* for the list of available options.

Rockstar halo catalogs are saved to the directory associated the `output_dir` keyword provided to the *HaloCatalog*. The number of files for each catalog is equal to the number of writers. The catalog files have the naming convention `halos_<catalog_number>.<processor_number>.bin`, where catalog number 0 is the first halo catalog calculated. For more information on loading these with yt, see Rockstar.

Parallelism

All three halo finders can be run in parallel using `mpirun` and by adding `yt.enable_parallelism()` to the top of the script. The computational domain will be divided evenly among all processes (among the writers in the case of Rockstar) with a small amount of padding to ensure halos on sub-volume boundaries are not split. For FoF and HOP, the number of processors used only needs to provided to `mpirun` (e.g., `mpirun -np 8` to run on 8 processors).

```
import yt

yt.enable_parallelism()
from yt.extensions.astro_analysis.halo_analysis import HaloCatalog

data_ds = yt.load("Enzo_64/RD0006/RedshiftOutput0006")
hc = HaloCatalog(
    data_ds=data_ds,
    finder_method="fof",
    finder_kwargs={"ptype": "stars", "padding": 0.02},
)
hc.create()
```

For more information on running `yt` in parallel, see Parallel Computation With yt.

Saving Halo Particles

As of version 1.1 of `yt_astro_analysis`, the ids of the particles belonging to each halo can be saved to the catalog when using either the *FoF* or *HOP* methods. The is enabled by default and can be disabled by setting `save_particles` to `False` in the `finder_kwargs` dictionary, as described above. Rockstar will also save halo particles to the `.bin` files. However, reading these is not currently supported in yt. See YTHaloCatalog for information on accessing halo particles for FoF and HOP catalogs.

3.1.2 Halo Analysis

Halo finding and analysis are combined into a single framework called the *HaloCatalog*. All halo catalogs created by the methods outlined in *Halo Finding* as well as those in the formats discussed in Halo Catalog Data can be loaded in to yt as first-class datasets. Once a halo catalog has been created, further analysis can be performed by providing both the halo catalog and the original simulation dataset to the *HaloCatalog*.

```
import yt
from yt.extensions.astro_analysis.halo_analysis import HaloCatalog

halos_ds = yt.load("rockstar_halos/halos_0.0.bin")
data_ds = yt.load("Enzo_64/RD0006/RedshiftOutput0006")
hc = HaloCatalog(data_ds=data_ds, halos_ds=halos_ds)
```

A data object can also be supplied via the keyword `data_source`, associated with either dataset, to control the spatial region in which halo analysis will be performed.

The *HaloCatalog* allows the user to create a pipeline of analysis actions that will be performed on all halos in the existing catalog. The analysis can be performed in parallel with separate processors or groups of processors being allocated to perform the entire pipeline on individual halos. The pipeline is setup by adding actions to the *HaloCatalog*. Each action is represented by a callback function that will be run on each halo. There are four types of actions:

- *Filters*
- *Quantities*
- *Callbacks*
- *Recipes*

A list of all available filters, quantities, and callbacks can be found in *Halo Analysis*. All interaction with this analysis can be performed by importing from halo_analysis.

Filters

A filter is a function that returns True or False. If the return value is True, any further queued analysis will proceed and the halo in question will be added to the final catalog. If the return value False, further analysis will not be performed and the halo will not be included in the final catalog.

An example of adding a filter:

```
hc.add_filter("quantity_value", "particle_mass", ">", 1e13, "Msun")
```

The two available filters are *quantity_value()* and *not_subhalo()*. More can be added by the user by defining a function that accepts a halo object as the first argument and then adding it as an available filter. If you think that your filter may be of use to the general community, you can add it to `yt_astro_analysis/halo_analysis/halo_catalog/halo_filters.py` and issue a pull request.

An example of defining your own filter:

```
def my_filter_function(halo):

    # Define condition for filter
    filter_value = True

    # Return a boolean value
    return filter_value
```

(continues on next page)

(continued from previous page)

```
# Add your filter to the filter registry
add_filter("my_filter", my_filter_function)

# ... Later on in your script
hc.add_filter("my_filter")
```

Quantities

A quantity is a call back that returns a value or values. The return values are stored within the halo object in a dictionary called "quantities." At the end of the analysis, all of these quantities will be written to disk as the final form of the generated halo catalog.

Quantities may be available in the initial fields found in the halo catalog, or calculated from a function after supplying a definition. An example definition of center of mass is shown below. If you think that your quantity may be of use to the general community, add it to `yt_astro_analysis/halo_analysis/halo_catalog/halo_quantities.py` and issue a pull request. Default halo quantities are:

- `particle_identifier` – Halo ID (e.g. 0 to N)
- `particle_mass` – Mass of halo
- `particle_position_x` – Location of halo
- `particle_position_y` – Location of halo
- `particle_position_z` – Location of halo
- `virial_radius` – Virial radius of halo

An example of adding a quantity:

```
hc.add_quantity("center_of_mass")
```

An example of defining your own quantity:

```
def my_quantity_function(halo):
    # Define quantity to return
    quantity = 5

    return quantity


# Add your filter to the filter registry
add_quantity("my_quantity", my_quantity_function)


# ... Later on in your script
hc.add_quantity("my_quantity")
```

This quantity will then be accessible for functions called later via the *quantities* dictionary that is associated with the halo object.

```
def my_new_function(halo):
    print(halo.quantities["my_quantity"])


add_callback("print_quantity", my_new_function)

# ... Anywhere after "my_quantity" has been called
hc.add_callback("print_quantity")
```

Callbacks

A callback is actually the super class for quantities and filters and is a general purpose function that does something, anything, to a Halo object. This can include hanging new attributes off the Halo object, performing analysis and writing to disk, etc. A callback does not return anything.

An example of using a pre-defined callback where we create a sphere for each halo with a radius that is twice the saved `radius`.

```
hc.add_callback("sphere", factor=2.0)
```

Currently available callbacks are located in `yt_astro_analysis/halo_analysis/halo_catalog/halo_callbacks.py`. New callbacks may be added by using the syntax shown below. If you think that your callback may be of use to the general community, add it to halo_callbacks.py and issue a pull request.

An example of defining your own callback:

```
def my_callback_function(halo):
    # Perform some callback actions here
    x = 2
    halo.x_val = x


# Add the callback to the callback registry
add_callback("my_callback", my_callback_function)


# ...  Later on in your script
hc.add_callback("my_callback")
```

Recipes

Recipes allow you to create analysis tasks that consist of a series of callbacks, quantities, and filters that are run in succession. An example of this is *`calculate_virial_quantities()`*, which calculates virial quantities by first creating a sphere container, performing 1D radial profiles, and then interpolating to get values at a specified threshold overdensity. All of these operations are separate callbacks, but the recipes allow you to add them to your analysis pipeline with one call. For example,

```
hc.add_recipe("calculate_virial_quantities", ["radius", "matter_mass"])
```

The available recipes are located in `yt_astro_analysis/halo_analysis/halo_catalog/halo_recipes.py`. New recipes can be created in the following manner:

```
def my_recipe(halo_catalog, fields, weight_field=None):
    # create a sphere
    halo_catalog.add_callback("sphere")
    # make profiles
    halo_catalog.add_callback("profile", ["radius"], fields, weight_field=weight_field)
    # save the profile data
    halo_catalog.add_callback("save_profiles", output_dir="profiles")


# add recipe to the registry of recipes
add_recipe("profile_and_save", my_recipe)


# ...  Later on in your script
hc.add_recipe("profile_and_save", ["density", "temperature"], weight_field="cell_mass")
```

Note, that unlike callback, filter, and quantity functions that take a `Halo` object as the first argument, recipe functions should take a `HaloCatalog` object as the first argument.

Running the Pipeline

After all callbacks, quantities, and filters have been added, the analysis begins with a call to *create()*.

```
hc.create()
```

The `save_halos` keyword determines whether the actual Halo objects are saved after analysis on them has completed or whether just the contents of their quantities dicts will be retained for creating the final catalog. The looping over halos uses a call to parallel_objects allowing the user to control how many processors work on each halo. The final catalog is written to disk in the output directory given when the *HaloCatalog* object was created.

All callbacks, quantities, and filters are stored in an actions list, meaning that they are executed in the same order in which they were added. This enables the use of simple, reusable, single action callbacks that depend on each other. This also prevents unnecessary computation by allowing the user to add filters at multiple stages to skip remaining analysis if it is not warranted.

Parallelism

Halo analysis using the *HaloCatalog* can be parallelized by adding `yt.enable_parallelism()` to the top of the script and running with `mpirun`.

```
import yt

yt.enable_parallelism()
from yt.extensions.astro_analysis.halo_analysis import HaloCatalog

halos_ds = yt.load("rockstar_halos/halos_0.0.bin")
data_ds = yt.load("Enzo_64/RD0006/RedshiftOutput0006")
hc = HaloCatalog(data_ds=data_ds, halos_ds=halos_ds)
hc.create(njobs="auto")
```

The nature of the parallelism can be configured with two keywords provided to the *create()* function: `njobs` and `dynamic`. If `dynamic` is set to False, halos will be distributed evenly over all processors. If `dynamic` is set to True, halos will be allocated to processors via a task queue. The `njobs` keyword determines the number of processor groups

over which the analysis will be divided. The default value for `njobs` is "auto". In this mode, a single processor will be allocated to analyze a halo. The `dynamic` keyword is overridden to False if the number of processors being used is even and True (use a task queue) if odd. Set `njobs` to -1 to mandate a single processor to analyze a halo and to a positive number to create that many processor groups for performing analysis. The number of processors used per halo will then be the total number of processors divided by `njobs`. For more information on running `yt` in parallel, see Parallel Computation With yt.

Loading Created Halo Catalogs

A *`HaloCatalog`* saved to disk can be reloaded as a yt dataset with the standard call to `load()`. See YTHaloCatalog for more information on loading a newly created catalog.

3.1.3 Overplotting Halo Annotations

The `yt_astro_analysis` package includes a function that allows one to overplot the locations of halos from a halo catalog on top of `yt` slices and projections. See Plot Modifications: Overplotting Contours, Velocities, Particles, and More for more information on the other available plot modifications.

To add the halo annotation to the set of available plot modifications, the following line must be added to your script.

```
import yt.extensions.astro_analysis.halo_analysis
```

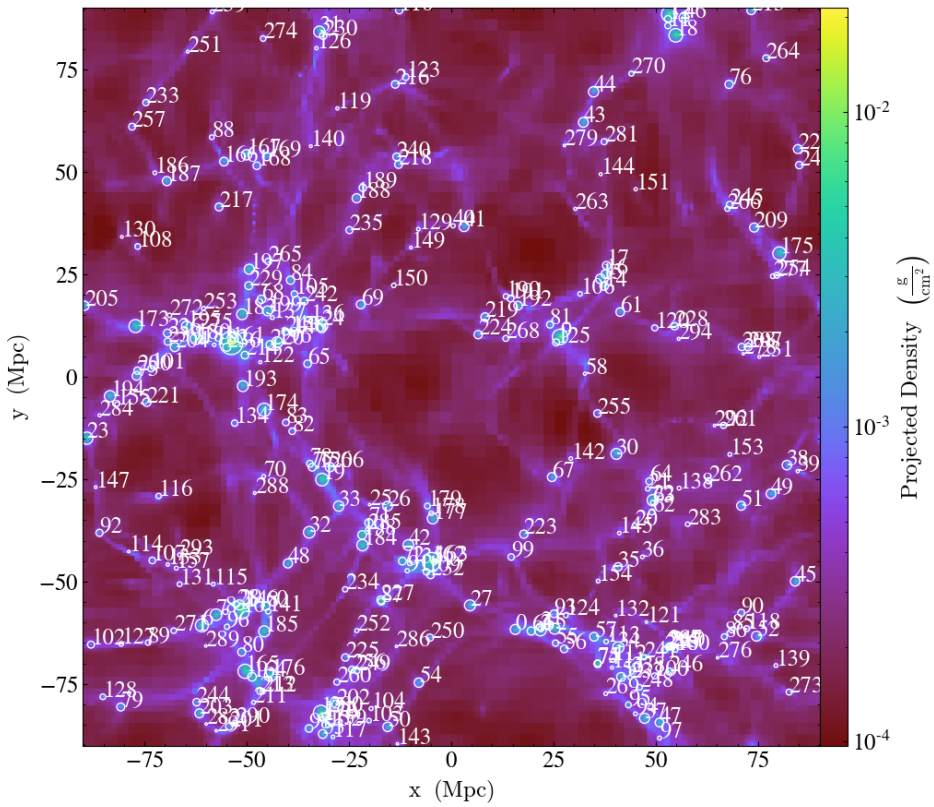
annotate_halos(*self*, *halo_catalog*, *circle_args=None*, *width=None*, *annotate_field=None*, *radius_field='virial_radius'*, *center_field_prefix='particle_position'*, *text_args=None*, *factor=1.0*)

This is a proxy for *`HaloCatalogCallback`*.

This will overplot circles denoting halo locations. The radius of the circle is given by the "virial_radius" field, but can be changed using the `radius_field` keyword argument. The user must provide one of the following:

1. a loaded `yt` `Dataset` of a halo catalog (e.g., a Rockstar catalog).
2. a `YTDataContainer` from a halo catalog dataset.
3. a *`HaloCatalog`*

```
import yt
import yt.extensions.astro_analysis.halo_analysis

data_ds = yt.load("Enzo_64/RD0006/RedshiftOutput0006")
halos_ds = yt.load("rockstar_halos/halos_0.0.bin")

p = yt.ProjectionPlot(data_ds, "z", ("gas", "density"))
p.annotate_halos(halos_ds, annotate_field="particle_identifier")
p.save()
```



3.1.4 Merger Trees

Merger trees can be created for *Rockstar-galaxies* halo catalogs using consistent-trees. The resulting merger tree data can be loaded with ytree. Note, halo finding must be done on a series of snapshots for this to work (see *Halo Finding on Multiple Snapshots*).

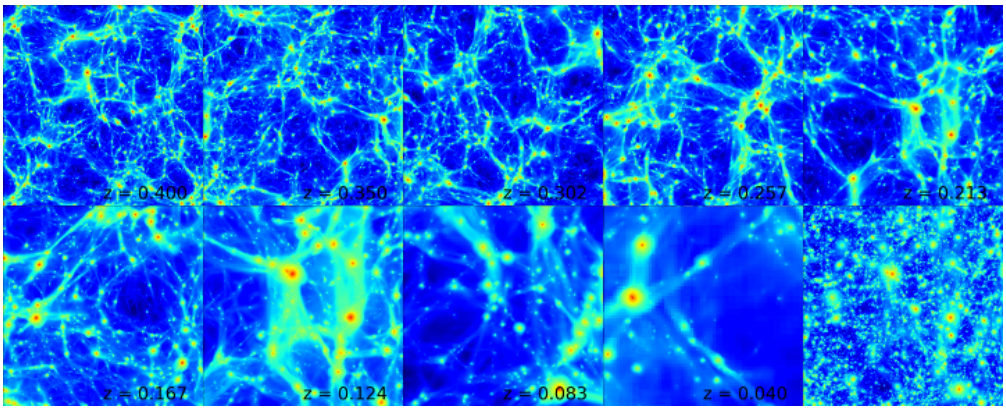
For halo catalogs created with *FoF* or *HOP*, the treefarm package can be used for generating merger trees. For this to work, member particles from halos must also be saved (see *Saving Halo Particles*). These merger trees can also be loaded with ytree.

3.2 Synthetic Observation

Methods for generating various types of synthetic observations from simulation data.

3.2.1 Light Cone Generator

Light cones are created by stacking multiple datasets together to continuously span a given redshift interval. To make a projection of a field through a light cone, the width of individual slices is adjusted such that each slice has the same angular size. Each slice is randomly shifted and projected along a random axis to ensure that the same structures are not sampled multiple times. A recipe for creating a simple light cone projection can be found in the cookbook under *Light Cone Projection*.



A light cone projection of the thermal Sunyaev-Zeldovich Y parameter from z = 0 to 0.4 with a 450x450 arcminute field of view using 9 individual slices. The panels shows the contributions from the 9 individual slices with the final light cone image shown in the bottom, right.

Configuring the Light Cone Generator

The required arguments to instantiate a `LightCone` object are the path to the simulation parameter file, the simulation type, the nearest redshift, and the furthest redshift of the light cone.

```
from yt.extensions.astro_analysis.cosmological_observation.api import LightCone

lc = LightCone("enzo_tiny_cosmology/32Mpc_32.enzo", "Enzo", 0.0, 0.1)
```

The additional keyword arguments are:

- `use_minimum_datasets` (*bool*): If True, the minimum number of datasets is used to connect the initial and final redshift. If False, the light cone solution will contain as many entries as possible within the redshift interval. Default: True.
- `deltaz_min` (*float*): Specifies the minimum Delta-z between consecutive datasets in the returned list. Default: 0.0.
- `minimum_coherent_box_fraction` (*float*): Used with `use_minimum_datasets` set to False, this parameter specifies the fraction of the total box size to be traversed before rerandomizing the projection axis and center. This was invented to allow light cones with thin slices to sample coherent large scale structure, but in practice does not work so well. Try setting this parameter to 1 and see what happens. Default: 0.0.
- `time_data` (*bool*): Whether or not to include time outputs when gathering datasets for time series. Default: True.

- `redshift_data` (*bool*): Whether or not to include redshift outputs when gathering datasets for time series. Default: True.
- `set_parameters` (*dict*): Dictionary of parameters to attach to ds.parameters. Default: None.
- **`output_dir` (*string*): The directory in which images and data files** will be written. Default: 'LC'.
- `output_prefix` (*string*): The prefix of all images and data files. Default: 'LightCone'.

Creating Light Cone Solutions

A light cone solution consists of a list of datasets spanning a redshift interval with a random orientation for each dataset. A new solution is calculated with the *`calculate_light_cone_solution()`* function:

```
lc.calculate_light_cone_solution(seed=123456789, filename="lightcone.dat")
```

The keyword argument are:

- `seed` (*int*): the seed for the random number generator. Any light cone solution can be reproduced by giving the same random seed. Default: None.
- `filename` (*str*): if given, a text file detailing the solution will be written out. Default: None.

Making a Light Cone Projection

With the light cone solution in place, projections with a given field of view and resolution can be made of any available field:

```
field = "density"
field_of_view = (600.0, "arcmin")
resolution = (60.0, "arcsec")
lc.project_light_cone(
    field_of_vew,
    resolution,
    field,
    weight_field=None,
    save_stack=True,
    save_slice_images=True,
)
```

The field of view and resolution can be specified either as a tuple of value and unit string or as a unitful `YTQuantity`. Additional keyword arguments:

- `weight_field` (*str*): the weight field of the projection. This has the same meaning as in standard projections. Default: None.
- `photon_field` (*bool*): if True, the projection data for each slice is decremented by 4 pi R ^{2} , where R is the luminosity distance between the observer and the slice redshift. Default: False.
- `save_stack` (*bool*): if True, the unflatted light cone data including each individual slice is written to an hdf5 file. Default: True.
- `save_final_image` (*bool*): if True, save an image of the final light cone projection. Default: True.
- `save_slice_images` (*bool*): save images for each individual projection slice. Default: False.
- `cmap_name` (*string*): color map for images. Default: "algae".

- `njobs` (*int*): The number of parallel jobs over which the light cone projection will be split. Choose -1 for one processor per individual projection and 1 to have all processors work together on each projection. Default: 1.
- `dynamic` (*bool*): If True, use dynamic load balancing to create the projections. Default: False.

Note: As of `yt-3.0`, the halo mask and unique light cone functionality no longer exist. These are still available in `yt-2.x`. If you would like to use these features in `yt-3.x`, help is needed to port them over. Contact the yt-users mailing list if you are interested in doing this.

3.2.2 Planning Simulations to use LightCones or LightRays

If you want to run a cosmological simulation that will have just enough data outputs to create a light cone or light ray, the *`plan_cosmology_splice()`* function will calculate a list of redshifts outputs that will minimally connect a redshift interval.

```
from yt.extensions.astro_analysis.cosmological_observation.api import CosmologySplice

my_splice = CosmologySplice("enzo_tiny_cosmology/32Mpc_32.enzo", "Enzo")
my_splice.plan_cosmology_splice(0.0, 0.1, filename="redshifts.out")
```

This will write out a file, formatted for simulation type, with a list of redshift dumps. The keyword arguments are:

- `decimals` (*int*): The decimal place to which the output redshift will be rounded. If the decimal place in question is nonzero, the redshift will be rounded up to ensure continuity of the splice. Default: 3.
- `filename` (*str*): If provided, a file will be written with the redshift outputs in the form in which they should be given in the enzo parameter file. Default: None.
- `start_index` (*int*): The index of the first redshift output. Default: 0.

3.2.3 Creating Position-Position-Velocity FITS Cubes

For an example of this functionality, see this sample notebook

The following routines have been moved to the Trident package.

- Light Ray Generator
- AbsorptionSpectrum

3.3 Exporting to External Radiation Transport Codes

3.3.1 Exporting to RADMC-3D

New in version 2.6.

RADMC-3D is a three-dimensional Monte-Carlo radiative transfer code that is capable of handling both line and continuum emission. yt comes equipped with a *`RadMC3DWriter`* class that exports AMR data to a format that RADMC-3D can read. Currently, only the ASCII-style data format is supported. In principle, this allows one to use RADMC-3D to make synthetic observations from any simulation data format that yt recognizes.

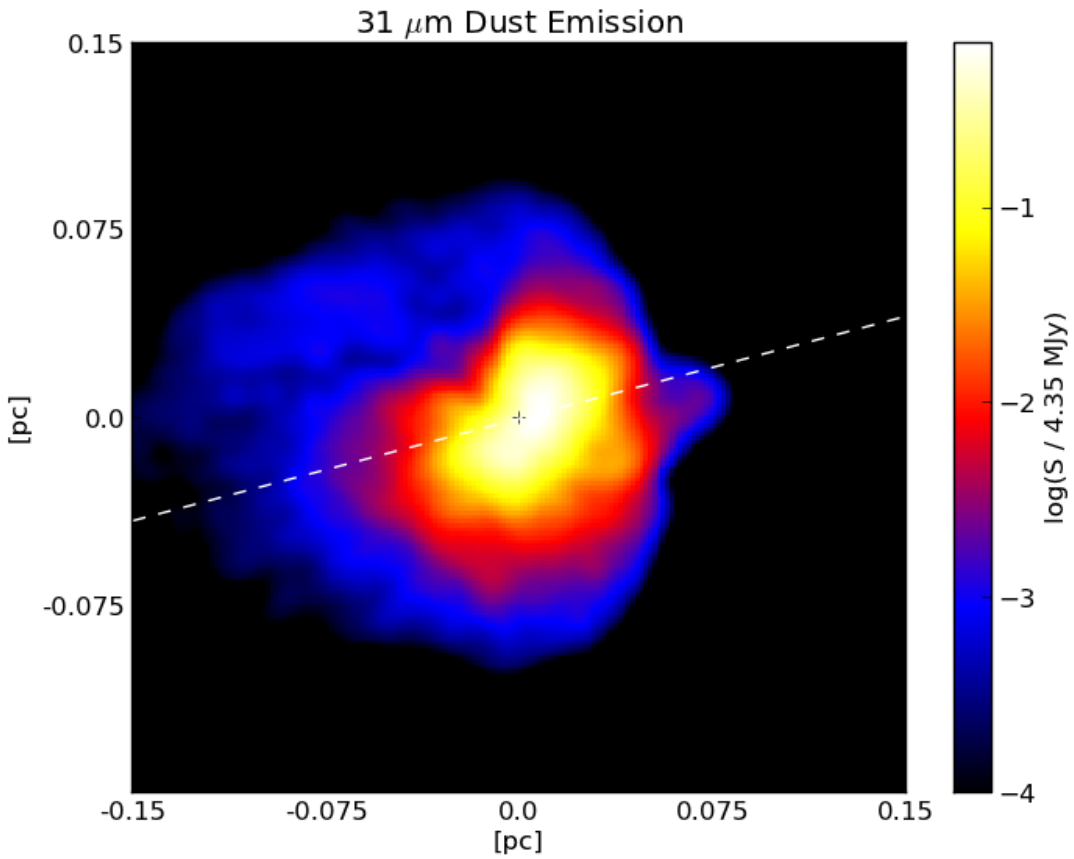


Fig. 1: Above: a sample image showing the continuum dust emission image around a massive protostar made using RADMC-3D and plotted with pyplot.

Continuum Emission

To compute thermal emission intensities, RADMC-3D needs several inputs files that describe the spatial distribution of the dust and photon sources. To create these files, first import the RADMC-3D exporter, which is not loaded into your environment by default:

```
import yt
import numpy as np
from yt.extensions.astro_analysis.radmc3d_export.api import RadMC3DWriter, RadMC3DSource
```

Next, load up a dataset and instantiate the *RadMC3DWriter*. For this example, we'll use the "StarParticle" dataset, available here.

```
ds = yt.load("StarParticles/plrd01000/")
writer = RadMC3DWriter(ds)
```

The first data file to create is the "amr_grid.inp" file, which describes the structure of the AMR index. To create this file, simply call:

```
writer.write_amr_grid()
```

Next, we must give RADMC-3D information about the dust density. To do this, we define a field that calculates the dust density in each cell. We assume a constant dust-to-gas mass ratio of 0.01:

```
dust_to_gas = 0.01


def _DustDensity(field, data):
    return dust_to_gas * data["density"]


ds.add_field(("gas", "dust_density"), function=_DustDensity, units="g/cm**3")
```

We save this information into a file called "dust_density.inp".

```
writer.write_dust_file(("gas", "dust_density"), "dust_density.inp")
```

Finally, we must give RADMC-3D information about any stellar sources that are present. To do this, we have provided the `RadMC3DSource` class. For this example, we place a single source with temperature 5780 K at the center of the domain:

```
radius_cm = 6.96e10
mass_g = 1.989e33
position_cm = [0.0, 0.0, 0.0]
temperature_K = 5780.0
star = RadMC3DSource(radius_cm, mass_g, position_cm, temperature_K)

sources_list = [star]
wavelengths_micron = np.logspace(-1.0, 4.0, 1000)

writer.write_source_files(sources_list, wavelengths_micron)
```

The last line creates the files "stars.inp" and "wavelength_micron.inp", which describe the locations and spectra of the stellar sources as well as the wavelengths RADMC-3D will use in it's calculations.

If everything goes correctly, after executing the above code, you should have the files "amr_grid.inp", "dust_density.inp", "stars.inp", and "wavelength_micron.inp" sitting in your working directory. RADMC-3D needs a few more configuration files to compute the thermal dust emission. In particular, you need an opacity file, like the "dustkappa_silicate.inp" file included in RADMC-3D, a main "radmc3d.inp" file that sets some runtime parameters, and a "dustopac.inp" that describes the assumed composition of the dust. yt cannot make these files for you; in the example that follows, we used a "radmc3d.inp" file that looked like:

```
nphot = 1000000
nphot_scat = 1000000
```

which basically tells RADMC-3D to use 1,000,000 photon packets instead of the default 100,000. The "dustopac.inp" file looked like:

```
2
1
----------------------------
1
0
silicate
----------------------------
```

To get RADMC-3D to compute the dust temperature, run the command:

```
./radmc3D mctherm
```

in the directory that contains your "amr_grid.inp", "dust_density.inp", "stars.inp", "wavelength_micron.inp", "radmc3d.inp", "dustkappa_silicate.inp", and "dustopac.inp" files. If everything goes correctly, you should get a "dust_temperature.dat" file in your working directory. Once that file is generated, you can use RADMC-3D to generate SEDs, images, and so forth. For example, to create an image at 31 microns, do the command:

```
./radmc3d image lambda 31 sizeau 30000 npix 800
```

which should create a file called "image.out". You can view this image using pyplot or whatever other plotting package you want. To facilitate this, we provide helper functions that parse the image.out file, returning a header dictionary with some useful metadata and an np.array containing the image values. To plot this image in pyplot, you could do something like:

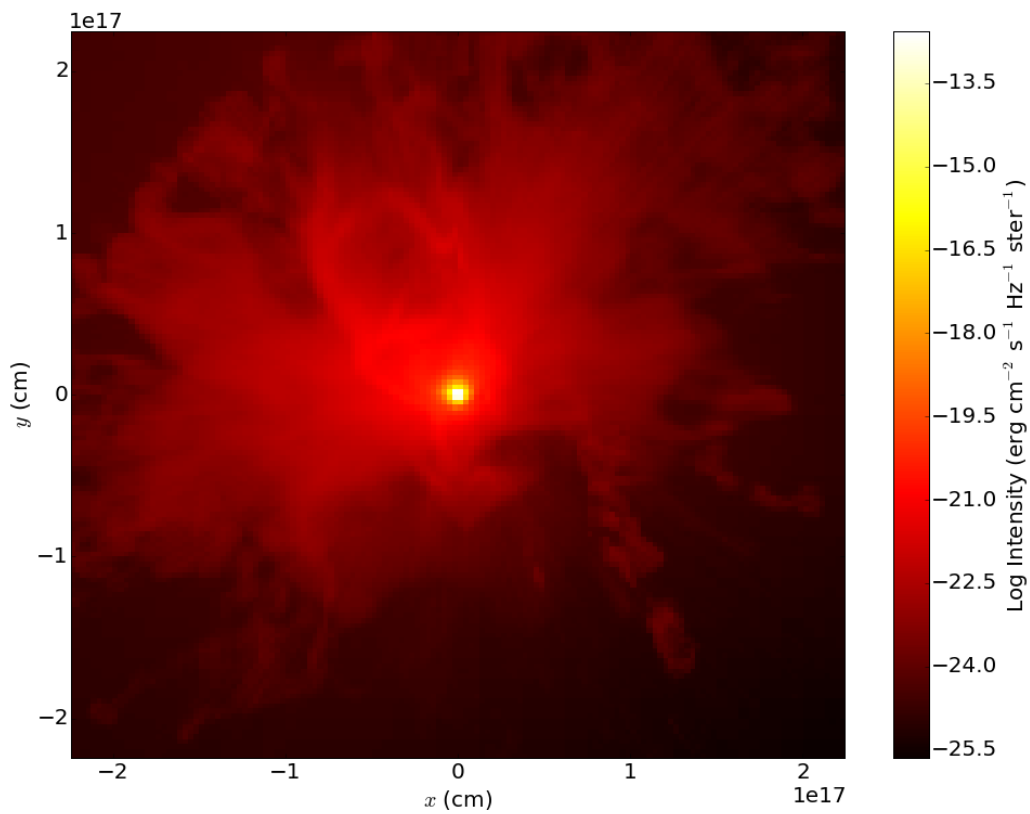
```
import matplotlib.pyplot as plt
import numpy as np
from yt.extensions.astro_analysis.radmc3d_export.api import read_radmc3d_image

header, image = read_radmc3d_image("image.out")

Nx = header["Nx"]
Ny = header["Ny"]

x_hi = 0.5 * header["pixel_size_cm_x"] * Nx
x_lo = -x_hi
y_hi = 0.5 * header["pixel_size_cm_y"] * Ny
y_lo = -y_hi

X = np.linspace(x_lo, x_hi, Nx)
Y = np.linspace(y_lo, y_hi, Ny)
```

(continues on next page)

(continued from previous page)

```
plt.pcolormesh(X, Y, np.log10(image), cmap="hot")
cbar = plt.colorbar()
plt.axis((x_lo, x_hi, y_lo, y_hi))
ax = plt.gca()
ax.set_xlabel(r"$x$ (cm)")
ax.set_ylabel(r"$y$ (cm)")
cbar.set_label(r"Log Intensity (erg cm$^{-2}$ s$^{-1}$ Hz$^{-1}$ ster$^{-1}$)")
plt.savefig("dust_continuum.png")
```

The resulting image should look like:



This barely scratches the surface of what you can do with RADMC-3D. Our goal here is just to describe how to use yt to export the data it knows about (densities, stellar sources, etc.) into a format that RADMC-3D can recognize.

Line Emission

The file format required for line emission is slightly different. The following script will generate two files, one called "numderdens_co.inp", which contains the number density of CO molecules for every cell in the index, and another called "gas-velocity.inp", which is useful if you want to include doppler broadening.

```
import yt
from yt.extensions.astro_analysis.radmc3d_export.api import RadMC3DWriter

x_co = 1.0e-4
mu_h = yt.YTQuantity(2.34e-24, "g")


def _NumberDensityCO(field, data):
    return (x_co / mu_h) * data["density"]


yt.add_field(("gas", "number_density_CO"), function=_NumberDensityCO, units="cm**-3")

ds = yt.load("IsolatedGalaxy/galaxy0030/galaxy0030")
writer = RadMC3DWriter(ds)

writer.write_amr_grid()
writer.write_line_file(("gas", "number_density_CO"), "numberdens_co.inp")
velocity_fields = ["velocity_x", "velocity_y", "velocity_z"]
writer.write_line_file(velocity_fields, "gas_velocity.inp")
```

CHAPTER

FOUR

THE COOKBOOK

These scripts and Jupyter notebooks provide detailed demonstrations for some of the functionality provide in the `yt_astro_analysis` package. All of the data used in these recipes is freely available here.

4.1 Example Scripts

4.1.1 Cosmological Analysis

These scripts demonstrate some basic and more advanced analysis that can be performed on cosmological simulation datasets.

Plotting Halos

This is a mechanism for plotting circles representing identified particle halos on an image. See *Halo Analysis* and *Overplotting Halo Annotations* for more information.

```
import yt
from yt.extensions.astro_analysis.halo_analysis.halo_catalog import HaloCatalog

# Load the dataset
ds = yt.load("Enzo_64/RD0006/RedshiftOutput0006")

# Load the halo list from a rockstar output for this dataset
halos = yt.load("rockstar_halos/halos_0.0.bin")

# Create the halo catalog from this halo list
hc = HaloCatalog(halos_ds=halos)
hc.load()

# Create a projection with the halos overplot on top
p = yt.ProjectionPlot(ds, "x", "density")
p.annotate_halos(hc)
p.save()
```

Light Cone Projection

This script creates a light cone projection, a synthetic observation that stacks together projections from multiple datasets to extend over a given redshift interval. See *Light Cone Generator* for more information.

```
import glob
import shutil

from yt.extensions.astro_analysis.cosmological_observation.api import LightCone

# Create a LightCone object extending from z = 0 to z = 0.1.

# We have already set up the redshift dumps to be
# used for this, so we will not use any of the time
# data dumps.
lc = LightCone(
    "enzo_tiny_cosmology/32Mpc_32.enzo",
    "Enzo",
    0.0,
    0.1,
    observer_redshift=0.0,
    time_data=False,
)

# Calculate a randomization of the solution.
lc.calculate_light_cone_solution(seed=123456789, filename="LC/solution.txt")

# Choose the field to be projected.
field = "szy"

# Use the LightCone object to make a projection with a 600 arcminute
# field of view and a resolution of 60 arcseconds.
# Set njobs to -1 to have one core work on each projection
# in parallel.
lc.project_light_cone(
    (600.0, "arcmin"),
    (60.0, "arcsec"),
    field,
    weight_field=None,
    save_stack=True,
    save_final_image=True,
    save_slice_images=True,
    njobs=-1,
)

# By default, the light cone projections are kept in the LC directory,
# but this moves them back to the current directory so that they're rendered
# in our cookbook.


for file in glob.glob("LC/*png"):
    shutil.move(file, ".")
```

4.2 Example Notebooks

- PPV Cube
- Sunyaev Zeldovich

CHAPTER

FIVE

CONTRIBUTING

We really want your contributions! As an official yt-project extension, everything in the yt Contributor Guide applies here.

This package uses the same development practices as yt itself, including code style and testing. As such, please consult the yt Developer Guide.

CHAPTER

SIX

CITING YT_ASTRO_ANALYSIS

If you use the yt_astro_analysis package for your work, please cite the yt_astro_analysis entry on zenodo.org as well as the yt method paper. Feel free to use the text below in your publications:

```
Analysis was performed using the yt_astro_analysis extension
(Smith et al. 2021) of the yt analysis toolkit (Turk et al. 2011).

Analysis was performed using the yt_astro_analysis extension
\citep{yt.astro.analysis} of the yt analysis toolkit \citep{yt}.
```

BbTeX entries are provided below:

```
@misc{yt.astro.analysis,
  author       = {Britton Smith and
                  Matthew Turk and
                  John ZuHone and
                  Nathan Goldbaum and
                  Cameron Hummels and
                  Hilary Egan and
                  John Wise and
                  Anthony Scopatz and
                  Miguel de Val-Borro and
                  Ben Keller and
                  Mark Richardson and
                  Clément Robert},
  title        = {yt_astro_analysis version 1.1.0},
  month        = dec,
  year         = 2021,
  doi          = {10.5281/zenodo.5783335},
  url          = {https://doi.org/10.5281/zenodo.5783335}
}

@ARTICLE{yt,
   author = {{Turk}, M.~J. and {Smith}, B.~D. and {Oishi}, J.~S. and {Skory}, S. and
{Skillman}, S.~W. and {Abel}, T. and {Norman}, M.~L.},
    title = "{yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data}",
  journal = {The Astrophysical Journal Supplement Series},
archivePrefix = "arXiv",
   eprint = {1011.3514},
 primaryClass = "astro-ph.IM",
 keywords = {cosmology: theory, methods: data analysis, methods: numerical},
     year = 2011,
```

(continues on next page)

(continued from previous page)

```
    month = jan,
   volume = 192,
      eid = {9},
    pages = {9},
      doi = {10.1088/0067-0049/192/1/9},
   adsurl = {http://adsabs.harvard.edu/abs/2011ApJS..192....9T},
  adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

CHAPTER

SEVEN

HELP

If you encounter problems, we want to help and there are lots of places to get help. As an extension of the yt project, we are members of the yt community. Any questions regarding ytree can be posted to the yt users list. You will also find interactive help on the yt slack channel.

CHAPTER

EIGHT

REFERENCE

Below are reference materials for the yt_astro_analysis package, including API documentation for all available functionality and a log of changes from each stable release.

8.1 API Reference

8.1.1 Halo Analysis

The `HaloCatalog` object is the primary means for performing custom analysis on cosmological halos. It is also the primary interface for halo finding.

HaloCatalog([halos_ds, data_ds, ...])	Create a HaloCatalog: an object that allows for the creation and association of data with a set of halo objects.
add_callback(name, function)	
add_filter(name, function)	
add_quantity(name, function)	
add_recipe(name, function)	
delete_attribute(halo, attribute)	Delete attribute from halo object.
halo_sphere(halo[, radius_field, factor, ...])	Create a sphere data container to associate with a halo.
iterative_center_of_mass(halo[, ...])	Adjust halo position by iteratively recalculating the center of mass while decreasing the radius.
load_profiles(halo[, storage, fields, ...])	Load profile data from disk.
phase_plot(halo[, output_dir, phase_args, ...])	Make a phase plot for the halo object.
profile(halo, bin_fields, profile_fields[, ...])	Create 1, 2, or 3D profiles of a halo.
save_profiles(halo[, storage, filename, ...])	Save profile data to disk.
sphere_bulk_velocity(halo)	Set the bulk velocity for the sphere.
sphere_field_max_recenter(halo, field)	Recenter the halo sphere on the location of the maximum of the given field.
virial_quantities(halo, fields[, ...])	Calculate the value of the given fields at the virial radius defined at the given critical density by interpolating from radial profiles.
not_subhalo(halo[, field_type])	Only return true if this halo is not a subhalo.
quantity_value(halo, field, operator, value, ...)	Filter based on a value in the halo quantities dictionary.

continues on next page

Table 1 – continued from previous page

calculate_virial_quantities(pipeline, fields)	Calculate virial quantities with the following procedure: 1.
HaloCatalogCallback(halo_catalog[, ...])	Plots circles at the locations of all the halos in a halo catalog with radii corresponding to the virial radius of each halo.

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog

class yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.**HaloCatalog**(*halos_ds=None*, *data_ds=None*, *data_source=None*, *halo_field_type='all'*, *finder_method=None*, *finder_kwargs=None*, *output_dir=None*)

Create a HaloCatalog: an object that allows for the creation and association of data with a set of halo objects.

A HaloCatalog object pairs a simulation dataset and the output from a halo finder, allowing the user to perform analysis on each of the halos found by the halo finder. Analysis is performed by providing callbacks: functions that accept a Halo object and perform independent analysis, return a quantity to be associated with the halo, or return True or False whether a halo meets various criteria. The resulting set of quantities associated with each halo is then written out to disk at a "halo catalog." This halo catalog can then be loaded in with yt as any other simulation dataset.

Parameters

- **halos_ds** (*str*) – Dataset created by a halo finder. If None, a halo finder should be provided with the finder_method keyword.
- **data_ds** (*str*) – Dataset created by a simulation.
- **data_source** (*data container*) – Data container associated with either the halos_ds to use for analysis. This can be used to restrict analysis to a subset of the full catalog. By default, the entire catalog will be analyzed.
- **halo_field_type** (*str*) – The field type for halos. This can be used to specify a certain type of halo in a dataset that contains multiple types. Default: "all"
- **finder_method** (*str*) – Halo finder to be used if no halos_ds is given.
- **output_dir** (*str*) – The top level directory into which analysis output will be written. Default: "halo_catalogs"
- **finder_kwargs** (*dict*) – Arguments to pass to the halo finder if finder_method is given.

Examples

```
>>> # create profiles or overdensity vs. radius for each halo and save to disk
>>> import yt
>>> from yt.extensions.astro_analysis.halo_analysis import HaloCatalog
>>> data_ds = yt.load("DD0064/DD0064")
>>> halos_ds = yt.load("rockstar_halos/halos_64.0.bin",
...                    output_dir="halo_catalogs/catalog_0064")
>>> hc = HaloCatalog(data_ds=data_ds, halos_ds=halos_ds)
>>> # filter out halos with mass < 1e13 Msun
>>> hc.add_filter("quantity_value", "particle_mass", ">", 1e13, "Msun")
>>> # create a sphere object with radius of 2 times the virial_radius field
>>> hc.add_callback("sphere", factor=2.0, radius_field="virial_radius")
>>> # make radial profiles
>>> hc.add_callback("profile", "radius", [("gas", "overdensity")],
...                 weight_field="cell_volume", accumulation=True)
>>> # save the profiles to disk
>>> hc.add_callback("save_profiles", output_dir="profiles")
>>> # create the catalog
>>> hc.create()
```

```
>>> # load in the saved halo catalog and all the profile data
>>> halos_ds = yt.load("halo_catalogs/catalog_0064/catalog_0064.0.h5")
>>> hc = HaloCatalog(halos_ds=halos_ds,
                     output_dir="halo_catalogs/catalog_0064")
>>> hc.add_callback("load_profiles", output_dir="profiles")
>>> hc.load()
```

See also:

add_callback, *add_filter*, *add_quantity*, *add_recipe*

__init__([halos_ds, data_ds, data_source, ...])	
add_callback(callback, *args, **kwargs)	Add a callback to the halo catalog action list.
add_filter(halo_filter, *args, **kwargs)	Add a filter to the halo catalog action list.
add_quantity(key, *args, **kwargs)	Add a quantity to the halo catalog action list.
add_recipe(recipe, *args, **kwargs)	Add a recipe to the halo catalog action list.
create([save_halos, save_output, njobs, dynamic])	Create the halo catalog given the callbacks, quantities, and filters that have been provided.
get_dependencies(fields)	
load([njobs, dynamic])	Load a previously created halo catalog.
partition_index_2d(axis)	
partition_index_3d(ds[, padding, rank_ratio])	
partition_index_3d_bisection_list()	Returns an array that is used to drive _partition_index_3d_bisection, below.
partition_region_3d(left_edge, right_edge[, ...])	Given a region, it subdivides it into smaller regions for parallel analysis.

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.__init__

HaloCatalog.**__init__**(*halos_ds=None*, *data_ds=None*, *data_source=None*, *halo_field_type='all'*, *finder_method=None*, *finder_kwargs=None*, *output_dir=None*)

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.add_callback

HaloCatalog.**add_callback**(*callback*, **args*, ***kwargs*)

Add a callback to the halo catalog action list.

A callback is a function that accepts and operates on a Halo object and does not return anything. Callbacks must exist within the callback_registry. Give additional args and kwargs to be passed to the callback here.

Parameters **callback** (*string*) – The name of the callback.

Examples

```
>>> # Here, a callback is defined and added to the registry.
>>> def _say_something(halo, message):
...     my_id = halo.quantities['particle_identifier']
...     print "Halo %d: here is a message - %s." % (my_id, message)
>>> add_callback("hello_world", _say_something)
```

```
>>> # Now this callback is accessible to the HaloCatalog object
>>> hc.add_callback("hello_world", "this is my message")
```

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.add_filter

HaloCatalog.**add_filter**(*halo_filter*, **args*, ***kwargs*)

Add a filter to the halo catalog action list.

A filter is a function that accepts a Halo object and returns either True or False. If True, any additional actions added to the list are carried out and the results are added to the final halo catalog. If False, any further actions are skipped and the halo will be omitted from the final catalog. Filters must exist within the filter_registry. Give additional args and kwargs to be passed to the filter function here.

Parameters **halo_filter** (*string*) – The name of the filter.

Examples

```
>>> # define a filter and add it to the register.
>>> def _my_filter(halo, mass_value):
...     return halo.quantities["particle_mass"] > unyt_quantity(mass_value, "Msun")
>>> # add it to the register
>>> add_filter("mass_filter", _my_filter)
```

```
>>> # add the filter to the halo catalog actions
>>> hc.add_filter("mass_value", 1e12)
```

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.add_quantity

`HaloCatalog.add_quantity(key, *args, **kwargs)`

Add a quantity to the halo catalog action list.

A quantity is a function that accepts a Halo object and return a value or values. These values are stored in a "quantities" dictionary associated with the Halo object. Quantities must exist within the quantity_registry. Give additional args and kwargs to be passed to the quantity function here.

Parameters

- **key** (*string*) – The name of the callback.
- **field_type** (*string*) – If not None, the quantity is the value of the field provided by the key parameter, taken from the halo finder dataset. This is the way one pulls values for the halo from the halo dataset. Default : None

Examples

```
>>> # pull the virial radius from the halo finder dataset
>>> hc.add_quantity("virial_radius", field_type="halos")
```

```
>>> # define a custom quantity and add it to the register
>>> def _mass_squared(halo):
...     # assume some entry "particle_mass" exists in the quantities dict
...     return halo.quantities["particle_mass"]**2
>>> add_quantity("mass_squared", _mass_squared)
```

```
>>> # add it to the halo catalog action list
>>> hc.add_quantity("mass_squared")
```

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.add_recipe

`HaloCatalog.add_recipe(recipe, *args, **kwargs)`

Add a recipe to the halo catalog action list.

A recipe is an operation consisting of a series of callbacks, quantities, and/or filters called in succession. Recipes can be used to store a more complex series of analysis tasks as a single entity.

Currently, the available recipe is `calculate_virial_quantities`.

Parameters **halo_recipe** (*string*) – The name of the recipe.

Examples

```
>>> import yt
>>> from yt.extensions.astro_analysis.halo_analysis import HaloCatalog
>>>
>>> data_ds = yt.load('Enzo_64/RD0006/RedshiftOutput0006')
>>> halos_ds = yt.load('rockstar_halos/halos_0.0.bin')
>>> hc = HaloCatalog(data_ds=data_ds, halos_ds=halos_ds)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # Filter out less massive halos
>>> hc.add_filter("quantity_value", "particle_mass", ">", 1e14, "Msun")
>>>
>>> # Calculate virial radii
>>> hc.add_recipe("calculate_virial_quantities", ["radius", "matter_mass"])
>>>
>>> hc.create()
```

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.create

`HaloCatalog.create(save_halos=False, save_output=True, njobs='auto', dynamic=False)`

Create the halo catalog given the callbacks, quantities, and filters that have been provided.

This is a wrapper around the main _run function with default arguments tuned for halo catalog creation. By default, halo objects are not saved but the halo catalog is written, opposite to the behavior of the load function.

Parameters

- **save_halos** (*bool*) – If True, a list of all Halo objects is retained under the "halo_list" attribute. If False, only the compiles quantities are saved under the "catalog" attribute. Default: False
- **save_output** (*bool*) – If True, save the final catalog to disk. Default: True
- **njobs** (*int*) – The number of jobs over which to divide halo analysis. If set to "auto", use a task queue if total number of processors is an odd number and divide jobs evenly if an even number. Default: "auto"
- **dynamic** (*int*) – If False, halo analysis is divided evenly between all available processors. If True, parallelism is performed via a task queue. If njobs is set to "auto", behavior is controlled in the way described above. Default: False

See also:

load

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.get_dependencies

`HaloCatalog.get_dependencies(fields)`

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.load

`HaloCatalog.load(njobs='auto', dynamic=False)`

Load a previously created halo catalog.

This is a wrapper around the main _run function with default arguments tuned for reloading halo catalogs and associated data. By default, halo objects are saved and the halo catalog is not written, opposite to the behavior of the create function.

Parameters

- **njobs** (*int*) – The number of jobs over which to divide halo analysis. If set to "auto", use a task queue if total number of processors is an odd number and divide jobs evenly if an even number. Default: "auto"

- **dynamic** (*int*) – If False, halo analysis is divided evenly between all available processors. If True, parallelism is performed via a task queue. If njobs is set to "auto", behavior is controlled in the way described above. Default: False

See also:

create

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.partition_index_2d

HaloCatalog.**partition_index_2d**(*axis*)

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.partition_index_3d

HaloCatalog.**partition_index_3d**(*ds*, *padding=0.0*, *rank_ratio=1*)

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.partition_index_3d_bisection_list

HaloCatalog.**partition_index_3d_bisection_list**()

Returns an array that is used to drive _partition_index_3d_bisection, below.

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.partition_region_3d

HaloCatalog.**partition_region_3d**(*left_edge*, *right_edge*, *padding=0.0*, *rank_ratio=1*)

Given a region, it subdivides it into smaller regions for parallel analysis.

comm
output_basename
output_dir
source_ds

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.comm

HaloCatalog.**comm = None**

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.output_basename

property HaloCatalog.**output_basename**

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.output_dir

property HaloCatalog.**output_dir**

yt_astro_analysis.halo_analysis.halo_catalog.halo_catalog.HaloCatalog.source_ds

property HaloCatalog.**source_ds**

yt_astro_analysis.halo_analysis.halo_catalog.analysis_operators.add_callback

yt_astro_analysis.halo_analysis.halo_catalog.analysis_operators.**add_callback**(*name*, *function*)

yt_astro_analysis.halo_analysis.halo_catalog.analysis_operators.add_filter

yt_astro_analysis.halo_analysis.halo_catalog.analysis_operators.**add_filter**(*name*, *function*)

yt_astro_analysis.halo_analysis.halo_catalog.analysis_operators.add_quantity

yt_astro_analysis.halo_analysis.halo_catalog.analysis_operators.**add_quantity**(*name*, *function*)

yt_astro_analysis.halo_analysis.halo_catalog.analysis_operators.add_recipe

yt_astro_analysis.halo_analysis.halo_catalog.analysis_operators.**add_recipe**(*name*, *function*)

yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.delete_attribute

yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.**delete_attribute**(*halo*, *attribute*)

Delete attribute from halo object.

Parameters

- **halo** (*Halo object*) – The Halo object to be provided by the HaloCatalog.
- **attribute** (*string*) – The attribute to be deleted.

yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.halo_sphere

yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.**halo_sphere**(*halo*, *radius_field='virial_radius'*, *factor=1.0*, *field_parameters=None*)

Create a sphere data container to associate with a halo.

Parameters

- **halo** (*Halo object*) – The Halo object to be provided by the HaloCatalog.

- **radius_field** (*string*) – Field to be retrieved from the quantities dictionary as the basis of the halo radius. Default: "virial_radius".
- **factor** (*float*) – Factor to be multiplied by the base radius for defining the radius of the sphere. Default: 1.0.
- **field_parameters** (*dict*) – Dictionary of field parameters to be set with the sphere created.

yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.iterative_center_of_mass

`yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.iterative_center_of_mass(halo, radius_field='virial_radiu inner_ratio=0.1, outer_radius=1.0, step_ratio=0.9, units='pc')`

Adjust halo position by iteratively recalculating the center of mass while decreasing the radius.

Parameters

- **halo** (*Halo object*) – The Halo object to be provided by the HaloCatalog.
- **radius_field** (*string*) – The halo quantity to be used as the radius for the sphere. Default: "virial_radius"
- **inner_ratio** (*float*) – The ratio of the smallest sphere radius used for calculating the center of mass to the initial radius. The sphere radius is reduced and center of mass recalculated until the sphere has reached this size. Default: 0.1
- **outer_radius** (*float*) – Sets the starting radius to begin testing the iterative technique in units of the radius field. When outer_radius starts out too large, sometimes the iterative algorithm wanders into a different halo. Default: 1.0
- **step_ratio** (*float*) – The multiplicative factor used to reduce the radius of the sphere after the center of mass is calculated. Default: 0.9
- **units** (*str*) – The units for printing out the distance between the initial and final centers. Default : "pc"

yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.load_profiles

`yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.load_profiles(halo, storage='profiles', fields=None, filename=None, output_dir='.')`

Load profile data from disk.

Parameters

- **halo** (*Halo object*) – The Halo object to be provided by the HaloCatalog.
- **storage** (*string*) – Name of the dictionary attribute to store profile data. Default: "profiles"

- **fields** (*string or list of strings*) – The fields to be loaded. If None, all fields present will be loaded. Default : None
- **filename** (*string*) – The name of the file to be loaded. The final filename will be "<filename>_<id>.h5". If None, filename is set to the value given by the storage keyword. Default: None
- **output_dir** (*string*) – Name of directory where profile data will be read. The full path will be the output_dir of the halo catalog concatenated with this directory. Default : "."

yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.phase_plot

`yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.phase_plot(halo, output_dir='.', phase_args=None, phase_kwargs=None)`

Make a phase plot for the halo object.

Parameters

- **halo** (*Halo object*) – The Halo object to be provided by the HaloCatalog.
- **output_dir** (*string*) – Name of directory where profile data will be written. The full path will be the output_dir of the halo catalog concatenated with this directory. Default : "."
- **phase_args** (*list*) – List of arguments to be given to PhasePlot.
- **phase_kwargs** (*dict*) – Dictionary of keyword arguments to be given to PhasePlot.

yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.profile

`yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.profile(halo, bin_fields, profile_fields, n_bins=32, extrema=None, logs=None, units=None, weight_field='cell_mass', accumulation=False, fractional=False, storage='profiles', output_dir='.')`

Create 1, 2, or 3D profiles of a halo.

Store profile data in a dictionary associated with the halo object.

Parameters

- **halo** (*Halo object*) – The Halo object to be provided by the HaloCatalog.
- **bin_fields** (*list of strings*) – The binning fields for the profile.
- **profile_fields** (*string or list of strings*) – The fields to be profiled.
- **n_bins** (*int or list of ints*) – The number of bins in each dimension. If None, 32 bins for each bin are used for each bin field. Default: 32.
- **extrema** (*dict of min, max tuples*) – Minimum and maximum values of the bin_fields for the profiles. The keys correspond to the field names. Defaults to the extrema of the bin_fields of the dataset. If a units dict is provided, extrema are understood to be in the units specified in the dictionary.

- **logs** (*dict of boolean values*) – Whether or not to log the bin_fields for the profiles. The keys correspond to the field names. Defaults to the take_log attribute of the field.
- **units** (*dict of strings*) – The units of the fields in the profiles, including the bin_fields.
- **weight_field** (*string*) – Weight field for profiling. Default : "cell_mass"
- **accumulation** (*bool or list of bools*) – If True, the profile values for a bin n are the cumulative sum of all the values from bin 0 to n. If -True, the sum is reversed so that the value for bin n is the cumulative sum from bin N (total bins) to n. If the profile is 2D or 3D, a list of values can be given to control the summation in each dimension independently. Default: False.
- **fractional** (*If True the profile values are divided by the sum of all*) – the profile data such that the profile represents a probability distribution function.
- **storage** (*string*) – Name of the dictionary to store profiles. Default: "profiles"
- **output_dir** (*string*) – Name of directory where profile data will be written. The full path will be the output_dir of the halo catalog concatenated with this directory. Default : "."

yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.save_profiles

`yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.`**`save_profiles`**(*halo*, *storage='profiles'*, *filename=None*, *output_dir='.'*)

Save profile data to disk.

Parameters

- **halo** (*Halo object*) – The Halo object to be provided by the HaloCatalog.
- **storage** (*string*) – Name of the dictionary attribute containing the profile data to be written. Default: "profiles"
- **filename** (*string*) – The name of the file to be written. The final filename will be "<filename>_<id>.h5". If None, filename is set to the value given by the storage keyword. Default: None
- **output_dir** (*string*) – Name of directory where profile data will be written. The full path will be the output_dir of the halo catalog concatenated with this directory. Default : "."

yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.sphere_bulk_velocity

`yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.`**`sphere_bulk_velocity`**(*halo*)

Set the bulk velocity for the sphere.

Parameters **halo** (*Halo object*) – The Halo object to be provided by the HaloCatalog.

yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.sphere_field_max_recenter

`yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.`**`sphere_field_max_recenter`**(*halo*, *field*)

Recenter the halo sphere on the location of the maximum of the given field.

Parameters

- **halo** (*Halo object*) – The Halo object to be provided by the HaloCatalog.
- **field** (*string*) – Field to be used for recentering.

yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.virial_quantities

`yt_astro_analysis.halo_analysis.halo_catalog.halo_callbacks.`**`virial_quantities`**(*halo*, *fields*, *overdensity_field=('gas', 'overdensity')*, *critical_overdensity=200*, *profile_storage='profiles'*)

Calculate the value of the given fields at the virial radius defined at the given critical density by interpolating from radial profiles.

Parameters

- **halo** (*Halo object*) – The Halo object to be provided by the HaloCatalog.
- **fields** (*string or list of strings*) – The fields whose virial values are to be calculated.
- **overdensity_field** (*string or tuple of strings*) – The field used as the overdensity from which interpolation is done to calculate virial quantities. Default: ("gas", "overdensity")
- **critical_overdensity** (*float*) – The value of the overdensity at which to evaluate the virial quantities. Overdensity is with respect to the critical density. Default: 200
- **profile_storage** (*string*) – Name of the halo attribute that holds the profiles to be used. Default: "profiles"

yt_astro_analysis.halo_analysis.halo_catalog.halo_filters.not_subhalo

`yt_astro_analysis.halo_analysis.halo_catalog.halo_filters.`**`not_subhalo`**(*halo*, *field_type='halos'*)

Only return true if this halo is not a subhalo.

This is used for halo finders such as Rockstar that output parent and subhalos together.

yt_astro_analysis.halo_analysis.halo_catalog.halo_filters.quantity_value

`yt_astro_analysis.halo_analysis.halo_catalog.halo_filters.quantity_value(halo, field, operator, value, units)`

Filter based on a value in the halo quantities dictionary.

Parameters

- **halo** (*Halo object*) – The Halo object to be provided by the HaloCatalog.
- **field** (*string*) – The field used for the evaluation.
- **operator** (*string*) – The comparison operator to be used ("<", "<=", "==", ">=", ">", etc.)
- **value** (*numneric*) – The value to be compared against.
- **units** (*string*) – Units of the value to be compared.

yt_astro_analysis.halo_analysis.halo_catalog.halo_recipes.calculate_virial_quantities

`yt_astro_analysis.halo_analysis.halo_catalog.halo_recipes.calculate_virial_quantities(pipeline, fields, weight_field=None, accumulation=True, radius_field='virial_radi factor=2.0, overdensity_field=('gas', 'overdensity'), critical_overdensity=200)`

Calculate virial quantities with the following procedure: 1. Create a sphere data container. 2. Create 1D radial profiles of overdensity and any requested fields. 3. Call virial_quantities callback to interpolate profiles for value of critical overdensity. 4. Delete profile and sphere objects from halo.

Parameters

- **halo** (*Halo object*) – The Halo object to be provided by the HaloCatalog.
- **fields** (*string or list of strings*) – The fields for which virial values are to be calculated.
- **weight_field** (*string*) – Weight field for profiling. Default : "cell_mass"
- **accumulation** (*bool or list of bools*) – If True, the profile values for a bin n are the cumulative sum of all the values from bin 0 to n. If -True, the sum is reversed so that the value for bin n is the cumulative sum from bin N (total bins) to n. If the profile is 2D or

3D, a list of values can be given to control the summation in each dimension independently. Default: False.

- **radius_field** (*string*) – Field to be retrieved from the quantities dictionary as the basis of the halo radius. Default: "virial_radius".
- **factor** (*float*) – Factor to be multiplied by the base radius for defining the radius of the sphere. Default: 2.0.
- **overdensity_field** (*string or tuple of strings*) – The field used as the overdensity from which interpolation is done to calculate virial quantities. Default: ("gas", "overdensity")
- **critical_overdensity** (*float*) – The value of the overdensity at which to evaluate the virial quantities. Overdensity is with respect to the critical density. Default: 200

yt_astro_analysis.halo_analysis.halo_catalog.plot_modifications.HaloCatalogCallback

class yt_astro_analysis.halo_analysis.halo_catalog.plot_modifications.**HaloCatalogCallback**(*halo_catalog, circle_args=None, circle_kwargs=None, width=None, annotate_field=None, radius_field='virial, center_field_prefix=', text_args=None, font_kwargs=Non, factor=1.0*)

Plots circles at the locations of all the halos in a halo catalog with radii corresponding to the virial radius of each halo.

Parameters

- **halo_catalog** (Dataset,) – YTDataContainer, or *HaloCatalog* The object containing halos to be overplotted. This can be a HaloCatalog object, a loaded halo catalog dataset, or a data container from a halo catalog dataset.
- **circle_args** (*list*) – Contains the arguments controlling the appearance of the circles, supplied to the Matplotlib patch Circle.
- **width** (*tuple*) – The width over which to select halos to plot, useful when overplotting to a slice plot. Accepts a tuple in the form (1.0, 'Mpc').
- **annotate_field** (*str*) – A field contained in the halo catalog to add text to the plot near the halo. Example: annotate_field = 'particle_mass' will write the halo mass next to each halo.
- **radius_field** (*str*) – A field contained in the halo catalog to set the radius of the circle which will surround each halo. Default: 'virial_radius'.
- **center_field_prefix** (*str*) – Accepts a field prefix which will be used to find the fields containing the coordinates of the center of each halo. Ex: 'particle_position' will result in

the fields 'particle_position_x' for x 'particle_position_y' for y, and 'particle_position_z' for z. Default: 'particle_position'.

- **text_args** (*dict*) – Contains the arguments controlling the text appearance of the annotated field.
- **factor** (*float*) – A number the virial radius is multiplied by for plotting the circles. Ex: factor = 2.0 will plot circles with twice the radius of each halo virial radius.

Examples

```
>>> import yt
>>> dds = yt.load("Enzo_64/DD0043/data0043")
>>> hds = yt.load("rockstar_halos/halos_0.0.bin")
>>> p = yt.ProjectionPlot(
...     dds, "x", ("gas", "density"), weight_field=("gas", "density")
... )
>>> p.annotate_halos(hds)
>>> p.save()
```

```
>>> # plot a subset of all halos
>>> import yt
>>> dds = yt.load("Enzo_64/DD0043/data0043")
>>> hds = yt.load("rockstar_halos/halos_0.0.bin")
>>> # make a region half the width of the box
>>> dregion = dds.box(
...     dds.domain_center - 0.25 * dds.domain_width,
...     dds.domain_center + 0.25 * dds.domain_width,
... )
>>> hregion = hds.box(
...     hds.domain_center - 0.25 * hds.domain_width,
...     hds.domain_center + 0.25 * hds.domain_width,
... )
>>> p = yt.ProjectionPlot(
...     dds,
...     "x",
...     ("gas", "density"),
...     weight_field=("gas", "density"),
...     data_source=dregion,
...     width=0.5,
... )
>>> p.annotate_halos(hregion)
>>> p.save()
```

```
>>> # plot halos from a HaloCatalog
>>> import yt
>>> from yt.extensions.astro_analysis.halo_analysis import HaloCatalog
>>> dds = yt.load("Enzo_64/DD0043/data0043")
>>> hds = yt.load("rockstar_halos/halos_0.0.bin")
>>> hc = HaloCatalog(data_ds=dds, halos_ds=hds)
>>> p = yt.ProjectionPlot(
...     dds, "x", ("gas", "density"), weight_field=("gas", "density")
... )
```

(continues on next page)

(continued from previous page)

```
>>> p.annotate_halos(hc)
>>> p.save()
```

__init__(halo_catalog[, circle_args, ...])	

yt_astro_analysis.halo_analysis.halo_catalog.plot_modifications.HaloCatalogCallback.__init__

`HaloCatalogCallback.__init__`(*halo_catalog*, *circle_args=None*, *circle_kwargs=None*, *width=None*, *annotate_field=None*, *radius_field='virial_radius'*, *center_field_prefix='particle_position'*, *text_args=None*, *font_kwargs=None*, *factor=1.0*)

region	

yt_astro_analysis.halo_analysis.halo_catalog.plot_modifications.HaloCatalogCallback.region

`HaloCatalogCallback.region = None`

8.1.2 Halo Finders

The halo finders should only be run from the `HaloCatalog`, but the links below display the various settings available for each.

FOFHaloFinder(ds[, subvolume, link, ...])	Friends-of-friends halo finder.
HOPHaloFinder(ds[, subvolume, threshold, ...])	HOP halo finder.
RockstarHaloFinder(ts[, num_readers, ...])	Spawns the Rockstar Halo finder, distributes particles and finds halos.

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.FOFHaloFinder

class `yt_astro_analysis.halo_analysis.halo_finding.halo_objects.FOFHaloFinder`(*ds*, *subvolume=None*, *link=0.2*, *dm_only=False*, *ptype='all'*, *padding=0.02*, *save_particles=True*)

Friends-of-friends halo finder.

Halos are found by linking together all pairs of particles closer than some distance from each other. Particles may have multiple links, and halos are found by recursively linking together all such pairs.

Larger linking lengths produce more halos, and the largest halos become larger. Also, halos become more filamentary and over-connected.

Davis et al. "The evolution of large-scale structure in a universe dominated by cold dark matter." ApJ (1985)

vol. 292 pp. 371-394

Parameters

- **ds** (*Dataset*) – The dataset on which halo finding will be conducted.
- **subvolume** (*yt.data_objects.data_containers.YTSelectionContainer*, optional) – A region over which HOP will be run, which can be used to run HOP on a subvolume of the full volume. Default = None, which defaults to the full volume automatically.
- **link** (*float*) – If positive, the interparticle distance (compared to the overall average) used to build the halos. If negative, this is taken to be the *actual* linking length, and no other calculations will be applied. Default = 0.2.
- **ptype** (*string*) – The type of particle to be used for halo finding. Default: 'all'.
- **padding** (*float*) – When run in parallel, the finder needs to surround each subvolume with duplicated particles for halo finidng to work. This number must be no smaller than the radius of the largest halo in the box in code units. Default = 0.02.
- **save_particles** (*bool*) – If True, output member particles for each halo. Default: True.

Examples

```
>>> import yt
>>> from yt.extensions.astro_analysis.halo_analysis import HaloCatalog
>>> data_ds = yt.load('Enzo_64/RD0006/RedshiftOutput0006')
>>> hc = HaloCatalog(data_ds=data_ds, finder_method='fof',
...                  finder_kwargs={"link": 0.2})
>>> hc.create()
```

__init__(ds[, subvolume, link, dm_only, ...])	Run hop on *data_source* with a given density *threshold*.
get_dependencies(fields)	
partition_index_2d(axis)	
partition_index_3d(ds[, padding, rank_ratio])	
partition_index_3d_bisection_list()	Returns an array that is used to drive _partition_index_3d_bisection, below.
partition_region_3d(left_edge, right_edge[, ...])	Given a region, it subdivides it into smaller regions for parallel analysis.

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.FOFHaloFinder.__init__

`FOFHaloFinder.__init__`(*ds*, *subvolume=None*, *link=0.2*, *dm_only=False*, *ptype='all'*, *padding=0.02*, *save_particles=True*)

Run hop on *data_source* with a given density *threshold*. Returns an iterable collection of *HopGroup* items.

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.FOFHaloFinder.get_dependencies

FOFHaloFinder.**get_dependencies**(*fields*)

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.FOFHaloFinder.partition_index_2d

FOFHaloFinder.**partition_index_2d**(*axis*)

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.FOFHaloFinder.partition_index_3d

FOFHaloFinder.**partition_index_3d**(*ds*, *padding=0.0*, *rank_ratio=1*)

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.FOFHaloFinder.partition_index_3d_bisection_list

FOFHaloFinder.**partition_index_3d_bisection_list**()

Returns an array that is used to drive _partition_index_3d_bisection, below.

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.FOFHaloFinder.partition_region_3d

FOFHaloFinder.**partition_region_3d**(*left_edge*, *right_edge*, *padding=0.0*, *rank_ratio=1*)

Given a region, it subdivides it into smaller regions for parallel analysis.

---

comm

---

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.FOFHaloFinder.comm

FOFHaloFinder.**comm = None**

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.HOPHaloFinder

class yt_astro_analysis.halo_analysis.halo_finding.halo_objects.**HOPHaloFinder**(*ds*, *subvolume=None*, *threshold=160*, *dm_only=False*, *ptype='all'*, *padding=0.02*, *total_mass=None*, *save_particles=True*)

HOP halo finder.

Halos are built by: 1. Calculating a density for each particle based on a smoothing kernel. 2. Recursively linking particles to other particles from lower density particles to higher. 3. Geometrically proximate chains are identified and 4. merged into final halos following merging rules.

Lower thresholds generally produce more halos, and the largest halos become larger. Also, halos become more filamentary and over-connected.

Eisenstein and Hut. "HOP: A New Group-Finding Algorithm for N-Body Simulations." ApJ (1998) vol. 498 pp. 137-142

Parameters

- **ds** (*Dataset*) – The dataset on which halo finding will be conducted.
- **subvolume** (*yt.data_objects.data_containers.YTSelectionContainer*, optional) – A region over which HOP will be run, which can be used to run HOP on a subvolume of the full volume. Default = None, which defaults to the full volume automatically.
- **threshold** (*float*) – The density threshold used when building halos. Default = 160.0.
- **ptype** (*string*) – The particle type to be used for halo finding. Default: 'all'.
- **padding** (*float*) – When run in parallel, the finder needs to surround each subvolume with duplicated particles for halo finidng to work. This number must be no smaller than the radius of the largest halo in the box in code units. Default = 0.02.
- **total_mass** (*float*) – If HOP is run on the same dataset mulitple times, the total mass of particles in Msun units in the full volume can be supplied here to save time. This must correspond to the particles being operated on, meaning if stars are included in the halo finding, they must be included in this mass as well, and visa-versa. If halo finding on a subvolume, this still corresponds with the mass in the entire volume. Default = None, which means the total mass is automatically calculated.
- **save_particles** (*bool*) – If True, output member particles for each halo. Default: True.

Examples

```
>>> import yt
>>> from yt.extensions.astro_analysis.halo_analysis import HaloCatalog
>>> data_ds = yt.load('Enzo_64/RD0006/RedshiftOutput0006')
>>> hc = HaloCatalog(data_ds=data_ds, finder_method='hop',
...                  finder_kwargs={"threshold": 160})
>>> hc.create()
```

__init__(ds[, subvolume, threshold, ...])	Run hop on *data_source* with a given density *threshold*.
get_dependencies(fields)	
partition_index_2d(axis)	
partition_index_3d(ds[, padding, rank_ratio])	
partition_index_3d_bisection_list()	Returns an array that is used to drive _partition_index_3d_bisection, below.
partition_region_3d(left_edge, right_edge[, ...])	Given a region, it subdivides it into smaller regions for parallel analysis.

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.HOPHaloFinder.__init__

HOPHaloFinder.**__init__**(*ds*, *subvolume=None*, *threshold=160*, *dm_only=False*, *ptype='all'*, *padding=0.02*, *total_mass=None*, *save_particles=True*)

Run hop on *data_source* with a given density *threshold*. Returns an iterable collection of *HopGroup* items.

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.HOPHaloFinder.get_dependencies

HOPHaloFinder.**get_dependencies**(*fields*)

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.HOPHaloFinder.partition_index_2d

HOPHaloFinder.**partition_index_2d**(*axis*)

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.HOPHaloFinder.partition_index_3d

HOPHaloFinder.**partition_index_3d**(*ds*, *padding=0.0*, *rank_ratio=1*)

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.HOPHaloFinder.partition_index_3d_bisection_list

HOPHaloFinder.**partition_index_3d_bisection_list**()

Returns an array that is used to drive _partition_index_3d_bisection, below.

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.HOPHaloFinder.partition_region_3d

HOPHaloFinder.**partition_region_3d**(*left_edge*, *right_edge*, *padding=0.0*, *rank_ratio=1*)

Given a region, it subdivides it into smaller regions for parallel analysis.

---

comm

---

yt_astro_analysis.halo_analysis.halo_finding.halo_objects.HOPHaloFinder.comm

HOPHaloFinder.**comm = None**

yt_astro_analysis.halo_analysis.halo_finding.rockstar.rockstar.RockstarHaloFinder

class yt_astro_analysis.halo_analysis.halo_finding.rockstar.rockstar.**RockstarHaloFinder**(*ts*, *num_readers=1*, *num_writers=None*, *outbase='rockstar_halo*, *particle_type='all'*, *mass_field='particle*, *star_types=None*, *force_res=None*, *initial_metric_scaling=*, *non_dm_metric_sca*, *suppress_galaxies=1*, *total_particles=None*, *dm_only=False*, *particle_mass=None*, *min_halo_size=25*, *restart=False*)

Spawns the Rockstar Halo finder, distributes particles and finds halos.

Rockstar has three main processes: reader, writer, and the server which coordinates reader/writer processes.

Parameters

- **ts** (*DatasetSeries, Dataset*) – The dataset or datsets on which halo finding will be run. If you intend to make a merger tree later, you must run Rockstar using a DatasetSeries containing all the snapshot to be included.
- **num_readers** (*int*) – The number of reader can be increased from the default of 1 in the event that a single snapshot is split among many files. This can help in cases where performance is IO-limited. Default is 1. If run inline, it is equal to the number of MPI threads.
- **num_writers** (*int*) – The number of writers determines the number of processing threads as well as the number of threads writing output data. The default is set to comm.size-num_readers-1. If run inline, the default is equal to the number of MPI threads.
- **outbase** (*str*) – This is where the out*list files that Rockstar makes should be placed. Default is 'rockstar_halos'.
- **particle_type** (*str*) – This is the "particle type" that can be found in the data. This can be a filtered particle or an inherent type.
- **mass_field** (*optional, str*) – The field to be used for the particle masses. The sampled field will be (<particle_type>, <mass_field>). This can be used to provide alternative particle masses for halo finding. Default: "particle_mass"
- **star_types** (*str list/array*) – The types (as returned by data((particle_type, particle_type)) to be recognized as star particles.
- **force_res** (*float*) – This parameter specifies the force resolution that Rockstar uses in units of Mpccm/h (comoving Mpc/h). If no value is provided, this parameter is automati-

cally set to the width of the smallest grid element in the simulation from the last data snapshot (i.e. the one where time has evolved the longest) in the time series: `ds_last.index.get_smallest_dx().to("Mpccm/h")`.

- **initial_metric_scaling** (*float*) – The position element of the fof distance metric is divided by this parameter, set to 1 by default. If the initial_metric_scaling=0.1 the position element will have 10 times more weight than the velocity element, biasing the metric towards position information more so than velocity information. That was found to be needed for hydro-ART simulations with 10's of parsecs resolution. Default: 1.0.
- **non_dm_metric_scaling** (*float*) – The metric scaling to be used for non-dm particles. The effect of this parameter is currently unknown. Default: 10.
- **suppress_galaxies** (*int*) – Wether to include non-dm halos (i.e. galaxies) in the catalogs. The effect of this parameter is currently unknown. Default: 1.
- **total_particles** (*int*) – If supplied, this is a pre-calculated total number of particles present in the simulation. For example, this is useful when analyzing a series of snapshots where the number of dark matter particles should not change and this will save some disk access time. If left unspecified, it will be calculated automatically. Default: `None`.
- **particle_mass** (optional, None, float, tuple, or `unyt_quantity`) – If supplied, this mass will be used to calculate the average particle spacing used in the friend-of-friends algorithm. The particle spacing will be (particle_mass / omega_matter * rho_cr)^1/3. If None, the mass is set as the minimum of all particles to be read. If a float, units are assumed to be in Msun/h. If a tuple, the format is assume to be (<value>, <units>). If a `unyt_quantity`, it must be convertible to units of Msun/h. To modify the masses of particles used for halo finding, see the mass_field keyword. Default: `None`.
- **restart** (*optional, bool*) – Set to True to have rockstar restart from the first uncompleted snapshot. If False, rockstar will start at the first snapshot in the simulation. Default: False

Returns

Return type None

Examples

To use the script below you must run it using MPI: *mpirun -np 4 python run_rockstar.py*

```
>>> import yt
>>> yt.enable_parallelism()
>>> from yt.extensions.astro_analysis.halo_analysis import HaloCatalog
>>> data_ds = yt.load('Enzo_64/RD0006/RedshiftOutput0006')
>>> hc = HaloCatalog(data_ds=data_ds, finder_method='rockstar',
...                  finder_kwargs={"num_readers": 1, "num_writers": 2})
>>> hc.create()
```

__init__(ts[, num_readers, num_writers, ...])

get_dependencies(fields)

partition_index_2d(axis)

continues on next page

Table 11 – continued from previous page

partition_index_3d(ds[, padding, rank_ratio])	
partition_index_3d_bisection_list()	Returns an array that is used to drive _partition_index_3d_bisection, below.
partition_region_3d(left_edge, right_edge[, ...])	Given a region, it subdivides it into smaller regions for parallel analysis.
run([block_ratio, callbacks, restart])	

yt_astro_analysis.halo_analysis.halo_finding.rockstar.rockstar.RockstarHaloFinder.__init__

RockstarHaloFinder.**__init__**(*ts*, *num_readers=1*, *num_writers=None*, *outbase='rockstar_halos'*, *particle_type='all'*, *mass_field='particle_mass'*, *star_types=None*, *force_res=None*, *initial_metric_scaling=1.0*, *non_dm_metric_scaling=10.0*, *suppress_galaxies=1*, *total_particles=None*, *dm_only=False*, *particle_mass=None*, *min_halo_size=25*, *restart=False*)

yt_astro_analysis.halo_analysis.halo_finding.rockstar.rockstar.RockstarHaloFinder.get_dependencies

RockstarHaloFinder.**get_dependencies**(*fields*)

yt_astro_analysis.halo_analysis.halo_finding.rockstar.rockstar.RockstarHaloFinder.partition_index_2d

RockstarHaloFinder.**partition_index_2d**(*axis*)

yt_astro_analysis.halo_analysis.halo_finding.rockstar.rockstar.RockstarHaloFinder.partition_index_3d

RockstarHaloFinder.**partition_index_3d**(*ds*, *padding=0.0*, *rank_ratio=1*)

yt_astro_analysis.halo_analysis.halo_finding.rockstar.rockstar.RockstarHaloFinder.partition_index_3d_bisection_

RockstarHaloFinder.**partition_index_3d_bisection_list**()

Returns an array that is used to drive _partition_index_3d_bisection, below.

yt_astro_analysis.halo_analysis.halo_finding.rockstar.rockstar.RockstarHaloFinder.partition_region_3d

RockstarHaloFinder.**partition_region_3d**(*left_edge*, *right_edge*, *padding=0.0*, *rank_ratio=1*)

Given a region, it subdivides it into smaller regions for parallel analysis.

yt_astro_analysis.halo_analysis.halo_finding.rockstar.rockstar.RockstarHaloFinder.run

`RockstarHaloFinder.run(block_ratio=1, callbacks=None, restart=False)`

comm	

yt_astro_analysis.halo_analysis.halo_finding.rockstar.rockstar.RockstarHaloFinder.comm

`RockstarHaloFinder.comm = None`

8.1.3 Cosmology Observation

Light cone generation and simulation analysis. (See also *Light Cone Generator*.)

CosmologySplice(parameter_filename, ...[, ...])	Class for splicing together datasets to extend over a cosmological distance.
LightCone(parameter_filename, ...[, ...])	Initialize a LightCone object.

yt_astro_analysis.cosmological_observation.cosmology_splice.CosmologySplice

class `yt_astro_analysis.cosmological_observation.cosmology_splice.CosmologySplice(parameter_filename, simulation_type, find_outputs=False)`

Class for splicing together datasets to extend over a cosmological distance.

__init__(parameter_filename, simulation_type)	
create_cosmology_splice(near_redshift, ...)	Create list of datasets capable of spanning a redshift interval.
plan_cosmology_splice(near_redshift, ...[, ...])	Create imaginary list of redshift outputs to maximally span a redshift interval.

yt_astro_analysis.cosmological_observation.cosmology_splice.CosmologySplice.__init__

`CosmologySplice.__init__(parameter_filename, simulation_type, find_outputs=False)`

yt_astro_analysis.cosmological_observation.cosmology_splice.CosmologySplice.create_cosmology_splice

`CosmologySplice.create_cosmology_splice(near_redshift, far_redshift, minimal=True, max_box_fraction=1.0, deltaz_min=0.0, time_data=True, redshift_data=True)`

Create list of datasets capable of spanning a redshift interval.

For cosmological simulations, the physical width of the simulation box corresponds to some Delta z, which varies with redshift. Using this logic, one can stitch together a series of datasets to create a continuous volume or length element from one redshift to another. This method will return such a list

Parameters

- **near_redshift** (*float*) – The nearest (lowest) redshift in the cosmology splice list.
- **far_redshift** (*float*) – The furthest (highest) redshift in the cosmology splice list.
- **minimal** (*bool*) – If True, the minimum number of datasets is used to connect the initial and final redshift. If false, the list will contain as many entries as possible within the redshift interval. Default: True.
- **max_box_fraction** (*float*) – In terms of the size of the domain, the maximum length a light ray segment can be in order to span the redshift interval from one dataset to another. If using a zoom-in simulation, this parameter can be set to the length of the high resolution region so as to limit ray segments to that size. If the high resolution region is not cubical, the smallest side should be used. Default: 1.0 (the size of the box)
- **deltaz_min** (*float*) – Specifies the minimum delta z between consecutive datasets in the returned list. Default: 0.0.
- **time_data** (*bool*) – Whether or not to include time outputs when gathering datasets for time series. Default: True.
- **redshift_data** (*bool*) – Whether or not to include redshift outputs when gathering datasets for time series. Default: True.

Examples

```
>>> co = CosmologySplice("enzo_tiny_cosmology/32Mpc_32.enzo", "Enzo")
>>> cosmo = co.create_cosmology_splice(1.0, 0.0)
```

yt_astro_analysis.cosmological_observation.cosmology_splice.CosmologySplice.plan_cosmology_splice

`CosmologySplice.plan_cosmology_splice(near_redshift, far_redshift, max_box_fraction=1.0, decimals=3, filename=None, start_index=0)`

Create imaginary list of redshift outputs to maximally span a redshift interval.

If you want to run a cosmological simulation that will have just enough data outputs to create a cosmology splice, this method will calculate a list of redshifts outputs that will minimally connect a redshift interval.

Parameters

- **near_redshift** (*float*) – The nearest (lowest) redshift in the cosmology splice list.
- **far_redshift** (*float*) – The furthest (highest) redshift in the cosmology splice list.
- **max_box_fraction** (*float*) – In terms of the size of the domain, the maximum length a light ray segment can be in order to span the redshift interval from one dataset to another.

If using a zoom-in simulation, this parameter can be set to the length of the high resolution region so as to limit ray segments to that size. If the high resolution region is not cubical, the smallest side should be used. Default: 1.0 (the size of the box)

- **decimals** (*int*) – The decimal place to which the output redshift will be rounded. If the decimal place in question is nonzero, the redshift will be rounded up to ensure continuity of the splice. Default: 3.
- **filename** (*string*) – If provided, a file will be written with the redshift outputs in the form in which they should be given in the enzo dataset. Default: None.
- **start_index** (*int*) – The index of the first redshift output. Default: 0.

Examples

```
>>> from yt.extensions.astro_analysis.cosmological_observation.api import␣
↪CosmologySplice
>>> my_splice = CosmologySplice('enzo_tiny_cosmology/32Mpc_32.enzo', 'Enzo')
>>> my_splice.plan_cosmology_splice(0.0, 0.1, filename='redshifts.out')
```

yt_astro_analysis.cosmological_observation.light_cone.light_cone.LightCone

class yt_astro_analysis.cosmological_observation.light_cone.light_cone.**LightCone**(*parameter_filename, simulation_type, near_redshift, far_redshift, observer_redshift=0.0, use_minimum_datasets=True, deltaz_min=0.0, minimum_coherent_box_fraction=, time_data=True, redshift_data=True, find_outputs=False, set_parameters=None, output_dir='LC', output_prefix='LightCone'*)

Initialize a LightCone object.

Parameters

- **near_redshift** (*float*) – The near (lowest) redshift for the light cone.
- **far_redshift** (*float*) – The far (highest) redshift for the light cone.
- **observer_redshift** (*float*) – The redshift of the observer. Default: 0.0.
- **use_minimum_datasets** (*bool*) – If True, the minimum number of datasets is used to connect the initial and final redshift. If false, the light cone solution will contain as many entries as possible within the redshift interval. Default: True.

- **deltaz_min** (*float*) – Specifies the minimum Δz between consecutive datasets in the returned list. Default: 0.0.
- **minimum_coherent_box_fraction** (*float*) – Used with use_minimum_datasets set to False, this parameter specifies the fraction of the total box size to be traversed before rerandomizing the projection axis and center. This was invented to allow light cones with thin slices to sample coherent large scale structure, but in practice does not work so well. Try setting this parameter to 1 and see what happens. Default: 0.0.
- **time_data** (*bool*) – Whether or not to include time outputs when gathering datasets for time series. Default: True.
- **redshift_data** (*bool*) – Whether or not to include redshift outputs when gathering datasets for time series. Default: True.
- **find_outputs** (*bool*) – Whether or not to search for datasets in the current directory. Default: False.
- **set_parameters** (*dict*) – Dictionary of parameters to attach to ds.parameters. Default: None.
- **output_dir** (*string*) – The directory in which images and data files will be written. Default: "LC".
- **output_prefix** (*string*) – The prefix of all images and data files. Default: "LightCone".

__init__(parameter_filename, ...[, ...])	

calculate_light_cone_solution([seed, filename])	Create list of projections to be added together to make the light cone.
create_cosmology_splice(near_redshift, ...)	Create list of datasets capable of spanning a redshift interval.
plan_cosmology_splice(near_redshift, ...[, ...])	Create imaginary list of redshift outputs to maximally span a redshift interval.
project_light_cone(field_of_view, ...[, ...])	Create projections for light cone, then add them together.

yt_astro_analysis.cosmological_observation.light_cone.light_cone.LightCone.__init__

LightCone.**__init__**(*parameter_filename*, *simulation_type*, *near_redshift*, *far_redshift*, *observer_redshift=0.0*, *use_minimum_datasets=True*, *deltaz_min=0.0*, *minimum_coherent_box_fraction=0.0*, *time_data=True*, *redshift_data=True*, *find_outputs=False*, *set_parameters=None*, *output_dir='LC'*, *output_prefix='LightCone'*)

yt_astro_analysis.cosmological_observation.light_cone.light_cone.LightCone.calculate_light_cone_solution

LightCone.**calculate_light_cone_solution**(*seed=None*, *filename=None*)

Create list of projections to be added together to make the light cone.

Several sentences providing an extended description. Refer to variables using back-ticks, e.g. *var*.

Parameters

- **seed** (*int*) – The seed for the random number generator. Any light cone solution can be reproduced by giving the same random seed. Default: None (each solution will be distinct).
- **filename** (*string*) – If given, a text file detailing the solution will be written out. Default: None.

yt_astro_analysis.cosmological_observation.light_cone.light_cone.LightCone.create_cosmology_splice

`LightCone.create_cosmology_splice(near_redshift, far_redshift, minimal=True, max_box_fraction=1.0, deltaz_min=0.0, time_data=True, redshift_data=True)`

Create list of datasets capable of spanning a redshift interval.

For cosmological simulations, the physical width of the simulation box corresponds to some Delta z, which varies with redshift. Using this logic, one can stitch together a series of datasets to create a continuous volume or length element from one redshift to another. This method will return such a list

Parameters

- **near_redshift** (*float*) – The nearest (lowest) redshift in the cosmology splice list.
- **far_redshift** (*float*) – The furthest (highest) redshift in the cosmology splice list.
- **minimal** (*bool*) – If True, the minimum number of datasets is used to connect the initial and final redshift. If false, the list will contain as many entries as possible within the redshift interval. Default: True.
- **max_box_fraction** (*float*) – In terms of the size of the domain, the maximum length a light ray segment can be in order to span the redshift interval from one dataset to another. If using a zoom-in simulation, this parameter can be set to the length of the high resolution region so as to limit ray segments to that size. If the high resolution region is not cubical, the smallest side should be used. Default: 1.0 (the size of the box)
- **deltaz_min** (*float*) – Specifies the minimum delta z between consecutive datasets in the returned list. Default: 0.0.
- **time_data** (*bool*) – Whether or not to include time outputs when gathering datasets for time series. Default: True.
- **redshift_data** (*bool*) – Whether or not to include redshift outputs when gathering datasets for time series. Default: True.

Examples

```
>>> co = CosmologySplice("enzo_tiny_cosmology/32Mpc_32.enzo", "Enzo")
>>> cosmo = co.create_cosmology_splice(1.0, 0.0)
```

yt_astro_analysis.cosmological_observation.light_cone.light_cone.LightCone.plan_cosmology_splice

`LightCone.plan_cosmology_splice(near_redshift, far_redshift, max_box_fraction=1.0, decimals=3, filename=None, start_index=0)`

Create imaginary list of redshift outputs to maximally span a redshift interval.

If you want to run a cosmological simulation that will have just enough data outputs to create a cosmology splice, this method will calculate a list of redshifts outputs that will minimally connect a redshift interval.

Parameters

- **near_redshift** (*float*) – The nearest (lowest) redshift in the cosmology splice list.
- **far_redshift** (*float*) – The furthest (highest) redshift in the cosmology splice list.
- **max_box_fraction** (*float*) – In terms of the size of the domain, the maximum length a light ray segment can be in order to span the redshift interval from one dataset to another. If using a zoom-in simulation, this parameter can be set to the length of the high resolution

region so as to limit ray segments to that size. If the high resolution region is not cubical, the smallest side should be used. Default: 1.0 (the size of the box)

- **decimals** (*int*) – The decimal place to which the output redshift will be rounded. If the decimal place in question is nonzero, the redshift will be rounded up to ensure continuity of the splice. Default: 3.
- **filename** (*string*) – If provided, a file will be written with the redshift outputs in the form in which they should be given in the enzo dataset. Default: None.
- **start_index** (*int*) – The index of the first redshift output. Default: 0.

Examples

```
>>> from yt.extensions.astro_analysis.cosmological_observation.api import␣
↪CosmologySplice
>>> my_splice = CosmologySplice('enzo_tiny_cosmology/32Mpc_32.enzo', 'Enzo')
>>> my_splice.plan_cosmology_splice(0.0, 0.1, filename='redshifts.out')
```

yt_astro_analysis.cosmological_observation.light_cone.light_cone.LightCone.project_light_cone

`LightCone.project_light_cone`(*field_of_view*, *image_resolution*, *field*, *weight_field=None*, *photon_field=False*, *save_stack=True*, *save_final_image=True*, *save_slice_images=False*, *cmap_name=None*, *njobs=1*, *dynamic=False*)

Create projections for light cone, then add them together.

Parameters

- **field_of_view** (*YTQuantity or tuple of (float, str)*) – The field of view of the image and the units.
- **image_resolution** (*YTQuantity or tuple of (float, str)*) – The size of each image pixel and the units.
- **field** (*string*) – The projected field.
- **weight_field** (*string*) – the weight field of the projection. This has the same meaning as in standard projections. Default: None.
- **photon_field** (*bool*) – if True, the projection data for each slice is decremented by 4 Pi R^2`, where R is the luminosity distance between the observer and the slice redshift. Default: False.
- **save_stack** (*bool*) – if True, the light cone data including each individual slice is written to an hdf5 file. Default: True.
- **save_final_image** (*bool*) – if True, save an image of the final light cone projection. Default: True.
- **save_slice_images** (*bool*) – save images for each individual projection slice. Default: False.
- **cmap_name** (*string*) – color map for images. Default: your default colormap.
- **njobs** (*int*) – The number of parallel jobs over which the light cone projection will be split. Choose -1 for one processor per individual projection and 1 to have all processors work together on each projection. Default: 1.

- **dynamic** (*bool*) – If True, use dynamic load balancing to create the projections. Default: False.

8.1.4 RADMC-3D exporting

RadMC3DLayer(level, parent, unique_id, LE, ...)	This class represents an AMR "layer" of the style described in the radmc3d manual.
RadMC3DWriter(ds[, max_level])	This class provides a mechanism for writing out data files in a format readable by radmc3d.

yt_astro_analysis.radmc3d_export.RadMC3DInterface.RadMC3DLayer

class yt_astro_analysis.radmc3d_export.RadMC3DInterface.**RadMC3DLayer**(*level*, *parent*, *unique_id*, *LE*, *RE*, *dim*)

This class represents an AMR "layer" of the style described in the radmc3d manual. Unlike yt grids, layers may not have more than one parent, so level L grids will need to be split up if they straddle two or more level L - 1 grids.

__init__(level, parent, unique_id, LE, RE, dim)	
get_overlap_with(grid)	Returns the overlapping region between two Layers, or a layer and a grid.
overlaps(grid)	Returns whether or not this layer overlaps a given grid

yt_astro_analysis.radmc3d_export.RadMC3DInterface.RadMC3DLayer.__init__

RadMC3DLayer.**__init__**(*level*, *parent*, *unique_id*, *LE*, *RE*, *dim*)

yt_astro_analysis.radmc3d_export.RadMC3DInterface.RadMC3DLayer.get_overlap_with

RadMC3DLayer.**get_overlap_with**(*grid*)

Returns the overlapping region between two Layers, or a layer and a grid. RE < LE means in any direction means no overlap.

yt_astro_analysis.radmc3d_export.RadMC3DInterface.RadMC3DLayer.overlaps

RadMC3DLayer.**overlaps**(*grid*)

Returns whether or not this layer overlaps a given grid

yt_astro_analysis.radmc3d_export.RadMC3DInterface.RadMC3DWriter

class yt_astro_analysis.radmc3d_export.RadMC3DInterface.**RadMC3DWriter**(*ds*, *max_level=2*)

This class provides a mechanism for writing out data files in a format readable by radmc3d. Currently, only the ASCII, "Layer" style file format is supported. For more information please see the radmc3d manual at: http://www.ita.uni-heidelberg.de/~dullemond/software/radmc-3d

Parameters

- **ds** (*Dataset*) – This is the dataset object corresponding to the simulation output to be written out.
- **max_level** (*int*) – An int corresponding to the maximum number of levels of refinement to include in the output. Often, this does not need to be very large as information on very high levels is frequently unobservable. Default = 2.

Examples

This will create a field called "DustDensity" and write it out to the file "dust_density.inp" in a form readable by RadMC3D.

```
>>> import yt
>>> from yt.extensions.astro_analysis.radmc3d_export.api import RadMC3DWriter
```

```
>>> dust_to_gas = 0.01
>>> def _DustDensity(field, data):
...     return dust_to_gas*data["Density"]
>>> yt.add_field("DustDensity", function=_DustDensity)
```

```
>>> ds = yt.load("galaxy0030/galaxy0030")
```

```
>>> writer = RadMC3DWriter(ds)
>>> writer.write_amr_grid()
>>> writer.write_dust_file("DustDensity", "dust_density.inp")
```

—

This example will create a field called "NumberDensityCO" and write it out to the file "numberdens_co.inp". It will also write out information about the gas velocity to "gas_velocity.inp" so that this broadening may be included in the radiative transfer calculation by radmc3d:

```
>>> import yt
>>> from yt.extensions.astro_analysis.radmc3d_export.api import RadMC3DWriter
```

```
>>> x_co = 1.0e-4
>>> mu_h = yt.Quantity(2.34e-24, 'g')
>>> def _NumberDensityCO(field, data):
```

(continues on next page)

(continued from previous page)

```
...     return (x_co/mu_h)*data["Density"]
>>> yt.add_field("NumberDensityCO", function=_NumberDensityCO)
```

```
>>> ds = yt.load("galaxy0030/galaxy0030")
>>> writer = RadMC3DWriter(ds)
```

```
>>> writer.write_amr_grid()
>>> writer.write_line_file("NumberDensityCO", "numberdens_co.inp")
>>> velocity_fields = ["velocity_x", "velocity_y", "velocity_z"]
>>> writer.write_line_file(velocity_fields, "gas_velocity.inp")
```

__init__(ds[, max_level])	

write_amr_grid()	This routine writes the "amr_grid.inp" file that describes the mesh radmc3d will use.
write_dust_file(field, filename)	This method writes out fields in the format radmc3d needs to compute thermal dust emission.
write_line_file(field, filename)	This method writes out fields in the format radmc3d needs to compute line emission.
write_source_files(sources, wavelengths)	This function creates the stars.inp and wavelength_micron.inp files that RadMC3D uses for its dust continuum calculations.

yt_astro_analysis.radmc3d_export.RadMC3DInterface.RadMC3DWriter.__init__

RadMC3DWriter.**__init__**(*ds*, *max_level=2*)

yt_astro_analysis.radmc3d_export.RadMC3DInterface.RadMC3DWriter.write_amr_grid

RadMC3DWriter.**write_amr_grid**()

This routine writes the "amr_grid.inp" file that describes the mesh radmc3d will use.

yt_astro_analysis.radmc3d_export.RadMC3DInterface.RadMC3DWriter.write_dust_file

RadMC3DWriter.**write_dust_file**(*field*, *filename*)

This method writes out fields in the format radmc3d needs to compute thermal dust emission. In particular, if you have a field called "DustDensity", you can write out a dust_density.inp file.

Parameters

- **field** (*string*) – The name of the field to be written out
- **filename** (*string*) – The name of the file to write the data to. The filenames radmc3d expects for its various modes of operations are described in the radmc3d manual.

yt_astro_analysis.radmc3d_export.RadMC3DInterface.RadMC3DWriter.write_line_file

`RadMC3DWriter.write_line_file(field, filename)`

This method writes out fields in the format radmc3d needs to compute line emission.

Parameters

- **field** (*string or list of 3 strings*) – If a string, the name of the field to be written out. If a list, three fields that will be written to the file as a vector quantity.
- **filename** (*string*) – The name of the file to write the data to. The filenames radmc3d expects for its various modes of operation are described in the radmc3d manual.

yt_astro_analysis.radmc3d_export.RadMC3DInterface.RadMC3DWriter.write_source_files

`RadMC3DWriter.write_source_files(sources, wavelengths)`

This function creates the stars.inp and wavelength_micron.inp files that RadMC3D uses for its dust continuum calculations.

Parameters

- **sources** (*a list of RadMC3DSource objects*) – A list that contains all the sources you would like yt to create
- **wavelengths** (*np.array of float values*) – An array listing the wavelength points you would like to use the radiative transfer calculation

8.2 ChangeLog

This is a log of changes to yt_astro_analysis over its release history.

8.2.1 Contributors

The CREDITS file contains the most up-to-date list of everyone who has contributed to the yt_astro_analysis source code.

8.2.2 Version 1.1.1

Release date: *January 27, 2022*

Bugfixes

- Make sure to initialize index before checking particle types PR #127
- Fix broken example with halo plotting PR #132
- Make total particles a 64 bit integer PR #133
- Set output directory properly for rockstar halo finder PR #134

Full Changelog: https://github.com/yt-project/yt_astro_analysis/compare/yt_astro_analysis-1.1.0...yt_astro_analysis-1.1.1

8.2.3 Version 1.1

Release date: *December 9, 2021*

New Features

- The HaloCatalog has been significantly refactored PR #58, PR #62 with the following additional improvements:
  - the interface to the Rockstar halo finder is now compatible with the latest version of Rockstar Galaxies PR #55
  - all halo finders now support being run with time-series of datasets
  - halo particle ids now savable with FoF and HOP halo finders PR #52
  - looping over halos is done with io chunks instead of ds.all_data for a significant speedup and reduction in memory
  - Allow more flexibility for specifying rockstar particle mass PR #84
  - Add restart option for rockstar PR #82
  - Adding an outer_radius parameter to the iterative COM callback PR #34
- Remove the sunyaev_zeldovich analysis module. This is now ytsz. PR #79
- Drop support for python 3.6 PR #100, PR #101

Minor Enhancements and Bugfixes

- significant project management and ci improvements PR #89, PR #90, PR #91, PR #92, PR #96, PR #95, PR #97, PR #108, PR #109
- Add annotate_halos function PR #98
- only access particle_type field in rockstar if it exists and is needed PR #111
- fix light cone projections with weight fields PR #37
- Fix HaloCatalog progress bar PR #40
- clarify rockstar error message about using the wrong number of MPI processes PR #42, PR #113
- check derived_field_list for base fields PR #43
- allow cosmology splice from a single dataset PR #49
- Fix iterator PR #68
- Support new config file format PR #65
- Enable circleci testing PR #44
- Add max_box_fraction to plan_cosmology_splice PR #76
- Fix HaloCatalog output_dir PR #81
- remove deprecated dm_only keyword from halo finder PR #57
- update amr_grid.inp PR #77

Full Changelog

8.2.4 Version 1.0

Release date: *October 11, 2018*

This is initial stable release of yt_astro_analysis. Before this, all code in here was contained in the yt package's `analysis_modules` submodule. Version 1.0 of yt_astro_analysis is functionally identical to the `analysis_modules` from yt version 3.5.0.

CHAPTER

NINE

CITING YT_ASTRO_ANALYSIS

If you use the yt_astro_analysis package for your work, please cite the yt_astro_analysis entry on zenodo.org as well as the yt method paper. Feel free to use the text below in your publications:

```
Analysis was performed using the yt_astro_analysis extension
(Smith et al. 2021) of the yt analysis toolkit (Turk et al. 2011).

Analysis was performed using the yt_astro_analysis extension
\citep{yt.astro.analysis} of the yt analysis toolkit \citep{yt}.
```

BbTeX entries are provided below:

```
@misc{yt.astro.analysis,
  author       = {Britton Smith and
                  Matthew Turk and
                  John ZuHone and
                  Nathan Goldbaum and
                  Cameron Hummels and
                  Hilary Egan and
                  John Wise and
                  Anthony Scopatz and
                  Miguel de Val-Borro and
                  Ben Keller and
                  Mark Richardson and
                  Clément Robert},
  title        = {yt_astro_analysis version 1.1.0},
  month        = dec,
  year         = 2021,
  doi          = {10.5281/zenodo.5783335},
  url          = {https://doi.org/10.5281/zenodo.5783335}
}

@ARTICLE{yt,
   author = {{Turk}, M.~J. and {Smith}, B.~D. and {Oishi}, J.~S. and {Skory}, S. and
{Skillman}, S.~W. and {Abel}, T. and {Norman}, M.~L.},
    title = "{yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data}",
  journal = {The Astrophysical Journal Supplement Series},
archivePrefix = "arXiv",
   eprint = {1011.3514},
 primaryClass = "astro-ph.IM",
 keywords = {cosmology: theory, methods: data analysis, methods: numerical},
     year = 2011,
```

(continues on next page)

(continued from previous page)

```
    month = jan,
   volume = 192,
      eid = {9},
    pages = {9},
      doi = {10.1088/0067-0049/192/1/9},
   adsurl = {http://adsabs.harvard.edu/abs/2011ApJS..192....9T},
  adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

INDEX

Symbols

A

B

C

D

F

G

H

I

L

N

O

P

Q

R

S

V

W