ytree Documentation

Release 1.0

Britton Smith

November 02, 2016

Contents

1 Installation **3**

2 Using ytree **5**

2.1 Loading, Using, and Saving Merger-trees 5

2.2 Making Merger-trees from Gadget FoF/Subfind 7

3 Help **11**

4 Citing ytree **13**

5 Search **15**

ytree is a merger-tree code based on the yt analysis toolkit. ytree can create merger-trees from Gadget FoF/Subfind catalogs, either for all halos or for a specific set of halos. ytree is able to load in merger-tree from the following formats:

- consistent-trees
- Rockstar halo catalogs without consistent-trees
- merger-trees made with ytree

All formats can be saved with a universal format that can be reloaded with ytree. Individual trees for single halos can also be saved. Similar to yt, fields queried for halos or trees are returned with units.

CHAPTER 1

Installation

ytree's main dependency is yt. Once you have installed yt following the instructions here, ytree can be installed using pip.

```
pip install ytree
```

And that's it!

CHAPTER 2

Using ytree

2.1 Loading, Using, and Saving Merger-trees

The `Arbor` class is responsible for loading and providing access to merger-tree data. Below, we discuss how to load in data and what one can do with it.

2.1.1 Loading Merger-tree data

`ytree` can load merger-tree data from multiple sources using the *~ytree.arbor.load* command. This command will guess the correct format and read in the data appropriately. For examples of loading each format, see below.

Consistent Trees

The consistent-trees format is typically one or a few files with a naming convention like "tree_0_0_0.dat". To load these files, just give the filename

```
import ytree
a = ytree.load("tree_0_0_0.dat")
```

Rockstar Catalogs

Rockstar catalogs with the naming convention "out_*.list" will contain information on the descendent ID of each halo and can be loaded independently of consistent-trees. This can be useful when your simulation has very few halos, such as in a zoom-in simulation. To load in this format, simply provide the path to one of these files.

```
import ytree
a = ytree.load("rockstar_halos/out_0.list")
```

TreeFarm

Merger-trees created with the *TreeFarm* method can be loaded in by providing the path to one of the catalogs created during the calculation.

```
import ytree
a = ytree.load("all_halos/fof_subhalo_tab_016.0.hdf5.0.h5")
```

Arbor

Once merger-tree data has been loaded, it can be saved to a universal format. This can be loaded by providing the file created.

```
import ytree
a = ytree.load("arbor.h5")
```

2.1.2 Working with Merger-trees

Once merger-tree data has been loaded into an `Arbor`, the individual trees will be stored in a list in the `trees` attribute.

```
>>> import ytree
>>> a = ytree.load("tree_0_0_0.dat")
yt : [INFO     ] 2016-09-26 15:35:57,279 Loading tree data from tree_0_0_0.dat.
Loading trees: 100%|| 327/327 [00:00<00:00, 4602.07it/s]
yt : [INFO     ] 2016-09-26 15:35:57,666 Arbor contains 327 trees with 10405 total nodes.
>>> print (len(a.trees))
327
>>> print (a.trees[0])
TreeNode[0,0]
```

A `TreeNode` is one halo in a merger-tree. The numbers correspond to the halo ID and the level in the tree. Like with `yt` data containers, fields can be queried in dictionary fashion.

```
>>> my_tree = a.trees[0]
>>> print (my_tree["mvir"])
1.147e+13 Msun/h
>>> print (my_tree["redshift"])
0.0
>>> print (my_tree["position"])
[ 69.95449  60.33949  50.64586] Mpc/h
>>> print (my_tree["velocity"])
[ -789.51  1089.31  1089.31] km/s
```

A halo's ancestors are stored as a list in the `ancestors` attribute.

```
>>> print my_tree.ancestors
[TreeNode[1,0]]
```

Iterating over a Tree

The `twalk` function provides an iterator that allows you to loop over all halos in the tree. This will iterate over all ancestors in a recursive fashion.

```
>>> for my_node in my_tree.twalk():
...     print (my_node)
```

Accessing the Trunk of the Tree

The `line` function allows one to query fields for the main trunk of the tree. By default, the "main trunk" follows the most massive progenitor.

```
>>> print my_tree.line("mvir")
[  1.14700000e+13   1.20700000e+13   1.23700000e+13   1.23700000e+13, ...,
   6.64000000e+12   5.13100000e+12   3.32000000e+12   1.20700000e+12
   2.71600000e+12] Msun/h
```

The selection method used by the `line` function can be changed by calling the `set_selector` function on the `Arbor`. For information on creating new selection methods, see the example, *~tree.tree_node_selector.max_field_value*.

```
>>> a.set_selector("min_field_value", "mvir")
```

Similar to `twalk`, the `lwalk` function provide an iterator over the trunk of a tree.

```
>>> for my_node in my_tree.lwalk():
...     print (my_node)
```

Similar to the `line` function, the `tree` function provides field access to the whole tree. However, since this is for all ancestors, note that these are not necessarily in chronological order.

```
>>> print my_tree.tree("mvir")
```

2.1.3 Saving Arbors and Trees

`Arbors` of any type can be saved to a universal file format which can be reloaded in the *same way*.

```
>>> a.save_arbor("my_arbor.h5")
yt : [INFO     ] 2016-09-26 16:45:40,064 Saving field data to yt dataset: my_arbor.h5.
>>> a2 = ytree.load("my_arbor.h5")
Loading trees: 100%|| 327/327 [00:00<00:00, 1086.22it/s]
yt : [INFO     ] 2016-09-26 16:46:26,383 Arbor contains 327 trees with 10405 total nodes.
```

Individual trees can be saved and reloaded in the same manner.

```
>>> fn = my_tree.save_tree()
yt : [INFO     ] 2016-09-26 16:47:09,931 Saving field data to yt dataset: tree_0_0.h5.
>>> atree = ytree.load(fn)
Loading trees: 100%|| 1/1 [00:00<00:00, 669.38it/s]
yt : [INFO     ] 2016-09-26 16:47:32,441 Arbor contains 1 trees with 45 total nodes.
```

2.2 Making Merger-trees from Gadget FoF/Subfind

The ytree `TreeFarm` can compute merger-trees either for all halos, starting at the beginning of the simulation, or for specific halos, starting at the final output and moving backward. These two use-cases are covered separately. Halo catalogs must be in the form created by the Gadget FoF halo finder or Subfind substructure finder.

2.2.1 Computing a Full Merger-tree

`TreeFarm` accepts a `yt` time-series object over which the merger-tree will be computed.

```
import yt
import ytree

ts = yt.DatasetSeries("data/groups_*/*.0.hdf5")
```

```
my_tree = TreeFarm(ts)
my_tree.trace_descendents("Group", filename="all_halos/")
```

The first argument to `trace_descendents` specifies the type of halo object to use. This will typically be either "Group" for FoF groups or Subhalo for Subfind groups. This process will create a new halo catalogs with the additional field representing the descendent ID for each halo. These can be loaded using `yt` like any other catalogs. Once complete, the final merger-tree can be loaded into a *TreeFarm Arbor*.

2.2.2 Computing a Targeted Merger-tree

Computing a full merger-tree can be extremely expensive when the simulation is large. Instead, merger-trees can be created for specific halos in the final dataset, then working backward. Below is an example of computing the merger-tree for only the most massive halo.

```
import yt
import ytree

ds = yt.load("data/groups_025/fof_subhalo_tab_025.0.hdf5")
i_max = np.argmax(ds.r["Group", "particle_mass"])
my_id = ds.r["particle_identifier"][i_max]

ts = yt.DatasetSeries("data/groups_*/*.0.hdf5")
my_tree = TreeFarm(ts)
my_tree.trace_ancestors("Group", my_id, filename="my_halo/")
```

Just as above, the resulting catalogs can then be loaded into a *TreeFarm Arbor*.

2.2.3 Optimizing Merger-tree Creation

Computing merger-trees can often be an expensive task. Below are some tips for speeding up the process.

Running in Parallel

ytree uses the parallel capabilities of yt to divide up the halo ancestor/descendent search over multiple processors. In order to do this, yt must be set up to run in parallel. See here for instructions. Once this is done, a call to `yt.enable_parallelism()` must be added to your script.

```
import yt
yt.enable_parallelism()
import ytree

ts = yt.DatasetSeries("data/groups_*/*.0.hdf5")
my_tree = TreeFarm(ts)
my_tree.trace_descendents("Group", filename="all_halos/")
```

That script must then be run with mpirun.

```
mpirun -np 4 python my_script.py
```

Optimizing Halo Candidate Selection

Halo ancestors and descendents are typically found by comparing particle IDs between two halos. The method of selecting which halos should be compared can greatly affect performance. By default, `TreeFarm` will compare a

halo against all halos in the next dataset. This is both the most robust and slowest method of matching ancestors and descendents. A smarter method is to select candidate matches from only a region around the target halo. For example, `TreeFarm` can be configured to select halos from a sphere centered on the current halo.

```
my_tree = TreeFarm(ts)
my_tree.set_selector("sphere", "virial_radius", factor=5)
my_tree.trace_descendents("Group", filename="all_halos/")
```

In the above example, candidate halos will be selected from a sphere that is five times the value of the "virial_radius" field. While this will speed up the calculation, a match will not be found if the ancestor/descendent is outside of this region. Some experimentation is recommended to find the optimal balance between speed and robustness.

Currently, the "sphere" selector is the only other selection method implemented, although others can be created easily. For an example, see *~ytree.halo_selector.sphere_selector*.

Searching for Fewer Ancestors

When computing a merger-tree for specific halos (*Computing a Targeted Merger-tree*), you only be interested in the most massive or the few most massive progenitors. If this is the case, `TreeFarm` can be configured to end the ancestor search when these have been found, rather than searching for all possible progenitors.

The `set_ancestry_filter` function places a filter on which ancestors of any given halo will be returned and followed in successive rounds of the merger-tree process. The "most_massive" filter instructs the `TreeFarm` to only keep the most massive ancestor. This will greatly reduce the number of halos included in the merger-tree and, therefore, speed up the calculation considerably. For an example of how to create a new filter, see *~ytree.ancestry_filter.most_massive*.

The filtering will only occur after all candidates have been checked for ancestry. An additional operation an be added to end the ancestry search after certain criteria have been met. In the call to `set_ancestry_short` below, the ancestry search will end as soon as an ancestor with at least 50% of the mass of the target halo has been found. For an example of how to create a new function of this type, see *~ytree.ancestry_short.most_massive*.

```
ts = yt.DatasetSeries("data/groups_*/*.0.hdf5")
my_tree = TreeFarm(ts)
my_tree.trace_ancestors("Group", my_id, filename="my_halo/")
my_tree.set_ancestry_filter("most_massive")
my_tree.set_ancestry_short("above_mass_fraction", 0.5)
```

CHAPTER 3

Help

Since ytree is heavily based on yt, the best way to get help is by joining the yt users list. Feel free to post any questions or ideas for development.

CHAPTER 4

Citing ytree

If you use ytree in your work, please cite it as "ytree, written by Britton smith" with a footnote pointing to http://ytree.readthedocs.io.

CHAPTER 5

Search

- search