

---

# **yt Documentation**

***Release 1.5-beta***

**Matthew Turk**

October 08, 2009



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	History . . . . .	3
1.2	What yt is and is not . . . . .	3
1.3	What functionality does yt offer? . . . . .	3
1.4	How do I cite yt? . . . . .	5
<b>2</b>	<b>Getting the Code</b>	<b>7</b>
2.1	Installation . . . . .	7
2.2	Notes on Common Installation Locations . . . . .	8
2.3	Installing by Hand . . . . .	9
2.4	Starting up YT . . . . .	10
<b>3</b>	<b>Analysis Philosophy</b>	<b>11</b>
3.1	Design Goals . . . . .	11
3.2	Object Methodology . . . . .	12
3.3	Derived Fields and Derived Quantities . . . . .	13
<b>4</b>	<b>How to Use YT</b>	<b>15</b>
4.1	Quick Start Guide . . . . .	15
4.2	A Slightly Longer Introduction . . . . .	21
4.3	Command Line Tool . . . . .	24
4.4	Using and Manipulating Objects and Fields . . . . .	27
4.5	Examining and Manipulating Particles . . . . .	30
4.6	Creating Derived Fields . . . . .	32
4.7	Parallel Computation With YT . . . . .	34
4.8	How to Make Plots . . . . .	36
<b>5</b>	<b>Cookbook</b>	<b>39</b>
5.1	Simple slice . . . . .	39
5.2	Simple projection . . . . .	42
5.3	Aligned cutting plane . . . . .	45
5.4	Sum mass in sphere . . . . .	47
5.5	Simple phase . . . . .	48
5.6	Simple profile . . . . .	49
5.7	Simple radial profile . . . . .	50
5.8	Halo finding . . . . .	51
5.9	Arbitrary vectors on slice . . . . .	52
5.10	Contours on slice . . . . .	52
5.11	Velocity vectors on slice . . . . .	53

5.12	Average value . . . . .	55
5.13	Find clumps . . . . .	56
5.14	Global phase plots . . . . .	58
5.15	Halo mass info . . . . .	59
5.16	Multi width save . . . . .	60
5.17	Zoomin frames . . . . .	64
5.18	Overplot particles . . . . .	69
5.19	Multi plot . . . . .	70
5.20	Multi plot 3x2 . . . . .	72
5.21	Time series phase . . . . .	75
5.22	Time series quantity . . . . .	77
5.23	Extract fixed resolution data . . . . .	79
<b>6</b>	<b>Advanced yt Usage</b>	<b>81</b>
6.1	Derived Quantities . . . . .	81
6.2	Plot Modification Mechanisms . . . . .	82
6.3	The Plugin File . . . . .	84
6.4	Creating 3D Datatypes . . . . .	85
6.5	Debugging and Driving YT . . . . .	85
<b>7</b>	<b>Extensions</b>	<b>89</b>
7.1	Halo Finding . . . . .	89
7.2	HaloProfiler . . . . .	92
7.3	Analyzing an Entire Simulation . . . . .	96
<b>8</b>	<b>Contributing Code</b>	<b>99</b>
8.1	Bug Fixes . . . . .	99
8.2	Licensing . . . . .	99
8.3	Fields and Extensions . . . . .	99
8.4	Analysis Code and Examples . . . . .	99
<b>9</b>	<b>Asking for Help</b>	<b>101</b>
9.1	The Mailing List . . . . .	101
9.2	Installation Issues . . . . .	101
9.3	Vanilla Usage Issues . . . . .	102
9.4	Customization and Scripting Issues . . . . .	102
9.5	How To Report A Bug . . . . .	102
<b>10</b>	<b>FAQ</b>	<b>103</b>
10.1	Why Python? . . . . .	103
10.2	Where can I learn more about Python? . . . . .	103
10.3	Who works on yt? . . . . .	103
10.4	What's up with the names? . . . . .	103
10.5	Are there any restrictions on my use of yt? . . . . .	103
10.6	How do I know what the units returned are? . . . . .	104
10.7	What are all these .yt files? . . . . .	104
10.8	How can I help? . . . . .	104
10.9	Something has gone wrong. What do I do? . . . . .	104
10.10	How do I specify an axis? . . . . .	104
10.11	Where can I go for support? . . . . .	105
<b>11</b>	<b>yt Methods</b>	<b>107</b>
11.1	Introduction . . . . .	107
11.2	Analysis Requirements . . . . .	108
11.3	Community Engagement . . . . .	108

11.4	Data Analysis Layer . . . . .	109
11.5	Plotting and Visualization Layer . . . . .	119
11.6	Constraints of Scale . . . . .	119
11.7	Frontends and Interfaces . . . . .	120
11.8	Embedding <code>yt</code> Inside Enzo . . . . .	120
11.9	Generalization to Other AMR Codes . . . . .	121
11.10	Immersive Visualization with VTK . . . . .	121
11.11	Community Involvement . . . . .	122
11.12	Future Directions . . . . .	122
<b>12</b>	<b>API Documentation</b>	<b>125</b>
12.1	<code>yt.lagos</code> Native AMR Data Structures . . . . .	125
12.2	<code>yt.lagos</code> Physical and Derived Data Objects . . . . .	131
12.3	<code>yt.raven</code> Plotting and Plot Interfaces . . . . .	140
12.4	<code>yt.reason</code> GUI Methods and Objects . . . . .	146
12.5	Convenience Functions . . . . .	146
12.6	<code>yt.extensions</code> Extensions API . . . . .	149
12.7	<code>yt.fido</code> File Storage and Management . . . . .	151
12.8	<code>yt.lagos.ParallelTools</code> Parallel Helper Functions . . . . .	152
<b>13</b>	<b>ChangeLog</b>	<b>153</b>
13.1	Version 1.5 . . . . .	153
13.2	Version 1.0 . . . . .	154
<b>14</b>	<b>Indices and tables</b>	<b>155</b>
	<b>Bibliography</b>	<b>157</b>
	<b>Module Index</b>	<b>159</b>
	<b>Index</b>	<b>161</b>



yt is a general-purpose toolkit designed to analyze, manage and plot adaptive mesh refinement data. It has been designed from the ground-up to work natively with the [Enzo](#) code, but it also supports analysis of output from the Orion code. It runs both interactively and non-interactively, and has been designed to support as many operations as possible in parallel.

If you use yt in a paper, I highly encourage you to read about our policy on *free repository space* for analysis code!

Below you'll find the table of contents. There's a super-quick-start guide to interactive data analysis, a tour of the objects and methodology of yt, a short cookbook, a guide extending, and – perhaps most important of all – a guide to the classes and functions available!

For more information, please visit [our homepage](#) and for help, please see *Asking for Help*.





# INTRODUCTION

## 1.1 History

My name is Matthew Turk, and I am the primary author of yt. I designed and implemented it during the course of my graduate studies working with Prof. Tom Abel at Stanford University, under the auspices of the Department of Energy through the SLAC National Accelerator Center and, briefly, at Los Alamos National Lab. It has evolved from a simple data-reader and exporter into what I believe is a fully-featured toolkit for analysis and visualization of adaptive mesh refinement data.

yt was designed to be a completely Free (as in beer *and* as in freedom – “free and libre” as the saying goes) user-extensible framework for analyzing and visualizing adaptive mesh refinement data, currently working with both Enzo and Orion data. It relies on no proprietary software – although it can be and has been extended to interface with proprietary software and libraries – and has been designed from the ground up to enable users to be as immersed in the data as they desire.

yt is currently being developed by a team consisting of me, Britton Smith, Stephen Skory, David Collins and Jeff Oishi. All development is conducted in the open, accessible at <http://yt.enzotools.org/>.

## 1.2 What yt is and is not

In some sense, yt is also designed to be rather utilitarian. By virtue of the fact that it has been written in an interpreted language, it can be somewhat slower than purely C-based analysis codes, although I believe that to be mitigated by a cleverness of algorithms and a substantially improved development time for the user. Several of the most computationally intensive problems have been written in C, or rely exclusively on C-based numerical libraries.

The primary goal has been, and will continue to be, to present an interface to the user that enables selection and analysis of arbitrary subsets of data.

## 1.3 What functionality does yt offer?

yt has evolved substantially over the time of its development. Here is a non-comprehensive list of features:

- Data Objects
  - Arbitrary data objects (Spheres, cylinders, rectangular prisms, arbitrary index selection)
  - Covering grids (smoothed and raw) for automatic ghost-zone generation
  - Identification of topologically-connected sets in arbitrary fields
  - Projections, orthogonal slices, oblique slices

- Axially-aligned rays
  - Memory-conserving 1-, 2- and 3-D profiles of arbitrary fields and objects.
  - Halo-finding (HOP) algorithm with full particle information and sphere access
  - Nearly **all** operations can be conducted in parallel
- Data access
  - Arbitrary field definition
  - Derived quantities (average values, spin parameter, bulk velocity, etc)
  - Custom C- written HDF5 backend for packed and unpacked AMR, NumPy-based HDF4 backend
  - CGS units used everywhere
  - Per-user field and quantity plugins
- Plotting
  - Mathtext TeX-like text formatting
  - Slices, projections, oblique slices
  - Profiles and phase diagrams
  - Linked zooms, colormaps, and saving across multiple plots
  - Contours, vector plots, annotated boxes, grid boundary plot overlays.
  - Simple 3D plotting of phase plots and volume-rendered boxes via hooks into the S2PLOT library
- GUI
  - Linked zooming via slider
  - Interactive re-centering
  - Length scales in human-readable coordinates
  - Drawing of circles for generation of data objects and phase plots
  - Image saving
  - Arbitrary plots within the GUI namespace
  - Full interpreter access to data objects
  - Macros and other scripts able to be run from within the namespace
- Command-line tools
  - Zooming movies
  - Time-series movies
  - HOP Halo Finding
- Access to components
  - Monetary cost: **FREE**.
  - Source code availability: **FULL**.
  - Portability: **YES**.

## 1.4 How do I cite yt?

If you use some of the advanced features of yt and would like to cite it in a publication, you should feel free to cite the [Proceedings Paper](#) with the following BibTeX entry:

```
@InProceedings{SciPyProceedings_46,
  author = {Matthew Turk},
  title = {Analysis and Visualization of Multi-Scale Astrophysical
Simulations Using Python and NumPy},
  booktitle = {Proceedings of the 7th Python in Science Conference},
  pages = {46 - 50},
  address = {Pasadena, CA USA},
  year = {2008},
  editor = {Ga\"el Varoquaux and Travis Vaught and Jarrod Millman},
}
```



# GETTING THE CODE

## 2.1 Installation

**Warning:** At the present time, binary packages are not supplied. The success of the *installation script* has largely removed the need for binaries.

YT comes with a handy installation script. This script will download and install all the necessary dependencies – from Python to YT – and then provide you with some information about how to modify your environment variables to ensure it is loaded properly.

To get a copy of the YT install script for Linux and Unix machines, you can obtain it from the subversion repository.

```
$ svn export http://svn.enzotools.org/yt/branches/yt-1.5/doc/install_script.sh
```

If you're running on Mac OSX, there is a different install script to run.

```
$ svn export http://svn.enzotools.org/yt/branches/yt-1.5/doc/install_script_osx.sh
```

Typically these scripts can just be run directly, but inside there are several options. Specifically, they can optionally install wxPython for GUI support, ZLIB (typically a good idea on 64-bit systems) and Mercurial (a great idea if you use *the barn!*) These options are all explained in *Using the Installation Script*.

### 2.1.1 Using the Installation Script

**Note:** The installation script is now the preferred means of installing a full set of packages – but if you are comfortable with python, feel free to install the code yourself!

In the `doc/` directory in the yt source distribution, there is a script, `install_script.sh`, designed to set up a full installation of yt, along with all the necessary dependencies. You can run this script from within a checkout of yt or an expanded tarball. If you are running on Mac OSX, you should run `install_script_osx.sh` instead.

**Note:** For convenience, yt will be installed in 'develop' mode, which means any changes in the source directory will be included the next time you import yt!

There are several variables you can set inside this script.

**DEST\_DIR** This is the location to which all source code will be downloaded and resulting built libraries installed.

**HDF5\_DIR** If you wish to link against existing HDF5 (*shared*) libraries, put the root path to the installation here. Statically linked libraries will not work.

**INST\_WXPYTHON** This is a boolean, set to 0 or 1, that governs whether or not wxPython should be installed.

**INST\_ZLIB** This is a boolean, set to 0 or 1, that governs whether or not zlib should be installed.

**INST\_HG** This is a boolean, set to 0 or 1, that governs whether or not mercurial should be installed. This is useful if you want to use scripts from [the barn](#).

**YT\_DIR** If you've got a source checkout of YT somewhere else, point to it with this!

**Warning:** If you run into problems, particularly anything involving `-fPIC`, it is likely that there's a problem with static libraries. Try asking the installer script to install HDF5 and ZLIB.

## 2.2 Notes on Common Installation Locations

### 2.2.1 Ranger (TACC)

YT installs out of the box on Ranger using the installation script. Zlib must be built by the script. The `pgi` module must first be swapped out for the `gcc/4.3.2` module. This set of commands has been reported to work for this purpose:

```
$ module unload mvapich-devel
$ module swap pgi gcc
$ module load mvapich-devel
```

Furthermore, errors citing GLIBC following logging out and logging back in can usually be solved by swapping out `gcc` for `pgi` again.

### 2.2.2 Kraken (NICS)

YT installs out of the box on Kraken using the installation script. Zlib must be built by the script. Before you begin, you must also ensure that the GNU programming environment is being used:

```
$ module swap PrgEnv-pgi PrgEnv-gnu
```

If you are going to try to run `yt` on the compute nodes, be aware that – while it does work – it will take a bit of effort because the compute nodes run Compute Node Linux. As a result, all the libraries have to be compiled statically – including all of Python and `yt`!

Stephen Skory has written a guide to getting Python compiled and running on the compute nodes [on the wiki](#).

### 2.2.3 Verne (NICS)

YT installs out of the box on Verne using the installation script. Zlib must be built by the script. Before you begin, you must also ensure that the GNU programming environment is being used:

```
$ module swap PE-pgi PE-gnu
```

### 2.2.4 Orange (SLAC)

YT installs out of the box if you either have the KIPAC `gfortran` installation in your path or use the `NUMPY_ARGS="--fcompiler=fake"` option as in the script.

### 2.2.5 Red (SLAC)

YT installs out of the box if you use the `NUMPY_ARGS="--fcompiler=fake"` option as in the script.

### 2.2.6 Cobalt (NCSA)

YT installs out of the box on Cobalt using the installation script. However, Zlib and HDF5 must both be installed via the installation script or linking errors will ensue. The GCC module, as opposed to the Intel Compiler module, should be loaded, but this may not be a hard requirement.

### 2.2.7 OS X

OS X installation can be tricky. It is best to use the `install_script_osx.sh` file, which will download fresh Python packages along with all dependencies. The [Enthought Python Distribution](#) is also a means of obtaining all these dependencies; if you use EPD, you will have to set up the file `hdf5.cfg` to point to the correct HDF5 installation from EPD. Users have reported some degree of success. Future versions of YT will leverage packages included in the EPD.

## 2.3 Installing by Hand

If you've ever installed a python package by hand before, YT should be easy to install. You will need to install the prerequisites first. A driving factor in the development of yt over the months leading to release 1.5 has been the reduction of dependencies. To that extent, only a few packages are required for the base usage, and a GUI toolkit if you are going to use the graphical user interface, Reason.

- [Python](#), at least version 2.4, but preferably 2.5 or 2.6.
- [HDF5](#), the data storage backend used by Enzo and yt (if you can run Enzo, this is already installed!)
- [NumPy](#), the fast numerical backend for Python
- [Matplotlib](#), the plotting package
- [wxPython](#), the GUI toolkit (optional)

(If you are only interested in manipulating data without any graphical plotting or interfaces, you only need to install HDF5, NumPy, and Python!)

Instructions for installing these packages is, unfortunately, beyond the scope of this document. However, there are copious directions on how to do so elsewhere. You may also consider installing the [Enthought Python Distribution](#), which includes all of the necessary packages.

You'll need to create a file in the YT directory called `hdf5.cfg` which points at the base of your HDF5 installation tree – usually this will be something like `/usr/local/`. Underneath this directory YT will look for `include` and `lib` directories containing the HDF5 files.

Once these dependencies have been met, YT can be installed in the standard manner:

```
$ cd yt/  
$ python2.6 setup.py install --prefix=/some/where/
```

## 2.4 Starting up YT

‘Starting up YT’ is a bit of a misnomer – there are many entry points to data analysis with YT. The simplest possible way to access YT is with the *Command Line Tool*. You can try this out just by typing

```
$ yt
```

and following the help instructions!



# ANALYSIS PHILOSOPHY

*Section author: J. S. Oishi <[jsoishi@astro.berkeley.edu](mailto:jsoishi@astro.berkeley.edu)>*

There are many tools available for analysis and visualization of AMR data; there are many just for `enzo`. So why `yt`? Along the road to answering that question, we shall take a somewhat philosophical scenic route. For the more pragmatically minded, the answer is simple: what `yt` does not yet do, you can make it do so. This is not as glib as it may seem: it is in fact the main philosophical tennant that underlies `yt`. In this section, it is not our goal to show you just how much `yt` already does. Instead, we will discuss how it is that `yt` does anything at all. In doing so, we hope to give you a sense of whether or not `yt` will align with your science goals.

At its core, `yt` is not a set of scripts to visualize AMR data, nor is it a set of low-level routines that return a homo- or even heterogeneous set of gridded data to your favorite scientific programming language—though `yt` incorporates both of these things, if your favorite scientific language is python. Instead, `yt` provides a series of objects, some common AMR code structures (such as hierarchies and levels in a nested mesh) and some physical (a cylinder, cube, or sphere somewhere in the problem domain), that allow you to process AMR data in order to get at the fundamental underlying physics.

## 3.1 Design Goals

`yt` evolved naturally out of three design goals, though when Matt was busy writing it, he never really thought about them. Over time, it became clear that they are real and furthermore that they are important to understanding how to use `yt`. These three goals are directed analysis, repeatability, and data exploration.

### 3.1.1 Directed Analysis: Answering a Question

One of the main motivators for `yt` is to make it possible to sit down with a definite question about an AMR dataset and code up a script that will provide an answer to that question. Indeed much of its object-oriented nature can be viewed as a way perform operations on a data object. Given that AMR simulations are usually used to track some kind of structure formation, be it shocks, stars, or galaxies, the data object may not be the entire domain, but some region within it that is interesting. This data object in hand, `yt` makes it easy (if possible: some tasks `yt` can merely make *possible*) to manipulate that data in such a way to answer a question related to your research.

### 3.1.2 Repeatability

In any scientific analysis, being able to repeat the set of steps that prepared an answer or physical quantity is essential. To that end, much of the usage of `yt` is focused around running scripts, describing actions and plots programmatically. Being able to write a script or conducting a set of commands that will reproduce identical results is fundamentally important, and `yt` will attempt to make that easy. It's for this reason that the interactive features of `yt` are not always

as advanced as they might otherwise be. We are actively working on integrating the SAGE notebook system into `yt`, which our preliminary tests suggest is a nice compromise between interactivity and repeatability.

### 3.1.3 Exploration

However, it is the serendipitous nature of science that often finding the right question is not obvious at first. This is certainly true for astrophysical simulation, especially so for simulations of structure formation. What are we looking for, and how will we know when we find it?

Quite often, the best way forward is to explore the simulation data as freely as possible. Without the ability for spot-examination, serendipitous discovery or general wandering, the code would be simply a pipeline, rather than a general tool. The flexible extensibility of `yt`, that is, the ability to create new derived quantities easily, as well as the ability to extract and display data regions in a variety of ways allows for this exploration.

## 3.2 Object Methodology

`yt` follows a strong object-oriented methodology. There is no real global state of `yt`; all state is contained within objects that encapsulate an AMR code object or physical region.

### 3.2.1 Physical Objects vs Code Objects

The best way to think about doing things with `yt` is to think first of objects. The AMR code puts a number of objects on disk, and `yt` has a matching set of objects to mimic these closely as possible. Your code runs (hopefully) a simulacrum of the physical universe, and thus in order to make sense of the output data, `yt` provides a set of objects meant to mimic the kinds of physical regions and processes you are interested in. For example, in a simulation of star formation out of some larger structure (the cosmic dark matter web, a turbulent molecular cloud), you might be interested in a sphere one parsec in radius around the point of maximum density. In a simulation of an accretion disk, you might want a cylindrical region of 1000 AU in radius and 10 AU in height with its axial vector aligned with the net angular momentum vector, which may be arbitrary with respect to the simulation cardinal axes. These are physical objects, and `yt` has a set of these too. Finally, you may wish to reduce the data to produce some essential data that represent a specific process. These reductions are also objects, and they are included in `yt` as well.

Somewhat separate from this, but in the same spirit, are plots. In `yt`, plots are also objects that one can create, manipulate, and save. In the case of plots, however, you tell `yt` what you want to see, and it can fetch data from the appropriate source.

In list form,

**Code Objects** These are things that are on the disk that the AMR code knows about – things like grids, data dumps, the grid hierarchy and so on.

**Physical Objects** These are objects like spheres, rectangular prisms, slices, and so on. These are collections of related data arranged by physical properties, and they are not necessarily associated with a single code object.

**Reduced Objects** These are objects created by taking a set of data and reducing it into a smaller format, suitable for a specific purpose. Histograms, 1-D profiles, and averages are all members of this category.

**Plots** Plots are somewhat different than other objects, as they are neither physical nor code. Instead, the plotting interface accepts information about what you want to see, then goes and fetches what is necessary—from code, physical, and reduced objects as necessary.

### 3.3 Derived Fields and Derived Quantities

While the heart of `yt` is the large set of basic code, physical, reduced, and plot objects already developed, in a metaphorical sense, its ‘soul’ is the fact that any of the objects can be used as starting points for creating fields and quantities of your own devices. Derived quantities and derived fields are the physical objects `yt` creates from the primitive variables the AMR code stores. These may or may not be the so-called primitive variables of fluid dynamics (density, velocity, energy): they are whatever your AMR code writes to disk.

Derived quantities are those data products derived from these variables such that the total amount of returned data is *less* than the number of cells. Derived fields, on the other hand, return a field with *equal* numbers of cells and the same geometry as the primitive variables from which it was derived. For example, `yt` could compute the gravitational potential at every point in space reconstructed from the density field.

`yt` already includes a large number of both derived fields and quantities, but its real power is that it is easy to create your own. See [Creating Derived Fields](#) for detailed instructions on creating derived fields.



# HOW TO USE YT

There's a lot inside yt. This section is designed to give you an idea of what's there, what you can do with it, and how to think about the yt environment.

Contents:

## 4.1 Quick Start Guide

If you're impatient, like me, you probably just want to pull up some data and take a look at it. This guide will help you out!

### 4.1.1 Starting IPython

If you've used the installation script that comes with yt, you should have an isolated environment containing Python 2.5, Matplotlib, yt, IPython, and maybe wxPython. Be sure to finish up the instructions by *prepending* the `LD_LIBRARY_PATH`, `PATH` and `PYTHONPATH` environment variables with the output of the script.

If you've done that, go ahead and start up our interactive yt environment:

```
$ iyt
```

It should start you up in an interpreter, and the namespace will be populated with the stuff you need. Really, the command `iyt` just opens up IPython and loads up yt, with some special commands available for you.

You're all set, so let's move on to the next step – actually opening up your data!

### 4.1.2 Opening Your Data File

You'll need to know the location of the parameter file from the output you want to look at. Let's pretend, for the sake of argument, it's `/home/mturk/data/galaxy1200.dir/galaxy1200` and that we have all the right permissions. So let's open it, and see what the maximum density is.

**Note:** In IPython, you get filename completion! So hit tab and it'll guess at what you want to open.

```
In [1]: pf = load("/home/mturk/data/galaxy1200.dir/galaxy1200")
```

```
In [2]: v, c = pf.h.find_max("Density")
```

And then in the variable `v` we have the value of the most dense cell, and in `c` we have the location of that point.

### 4.1.3 Making Plots

But hey, what good is the data if we can't see it? So let's make some plots! First we need to get a `PlotCollectionInteractive` object, and then we'll add some slices and projections to it. Note that we use 0, 1, 2 to refer to 'x', 'y', 'z' axes.

```
In [3]: pc = PlotCollectionInteractive(pf)
In [4]: pc.add_slice("Temperature", 0)
yt.raven INFO      2008-10-25 11:42:58,429 Added slice of Temperature at x = 0.953125 with 'center' =
Out[4]: <yt.raven.PlotTypes.SlicePlot instance at 0x9882cec>
```

```
In [5]: pc.add_slice("Density", 0)
yt.raven INFO      2008-10-25 11:43:45,608 Added slice of Density at x = 0.953125 with 'center' =
Out[5]: <yt.raven.PlotTypes.SlicePlot instance at 0xab83eec>
```

A window should now pop up for each of these plots. One will be a line integral through the simulation, and the other will be a slice. (If you had used the `PlotCollection` object, they'd be created off-screen – this is the right way to make plots programmatically in scripts.)

We can also adjust the width of the plots very easily:

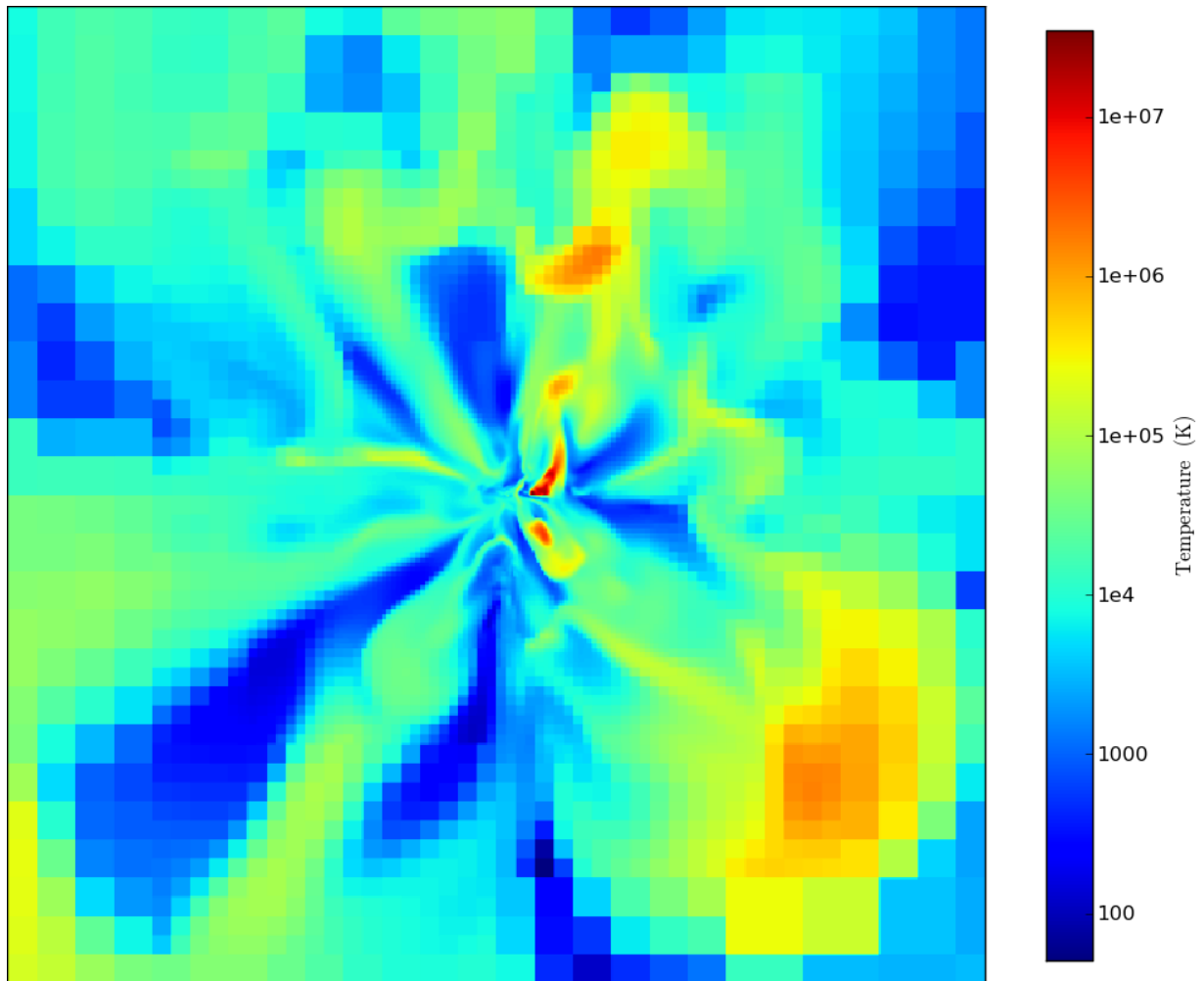
```
In [6]: pc.set_width(100, 'kpc')
```

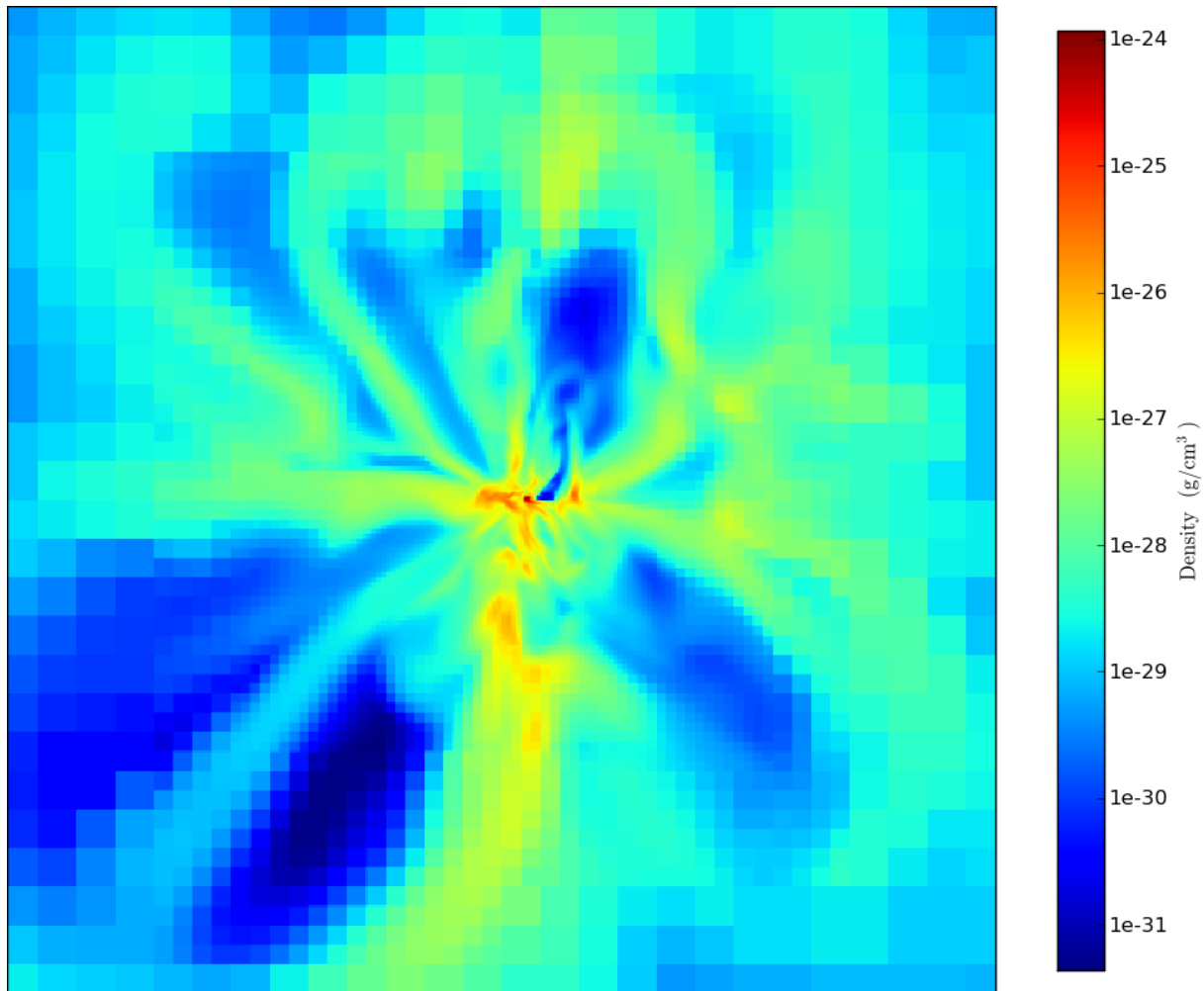
The center is set to the most dense location by default. (For more information, see the documentation for `PlotCollection`.)

### 4.1.4 Saving Plots

Even though the windows are open, we can save these to the file system at high resolution.

```
In [7]: pc.save()
Out[7]: ['galaxy1200_Slice_x_Temperature.png', 'galaxy1200_Slice_x_Density.png']
```





And that's it! The plots get saved out, and it returns to you a list of their filenames.

**Note:** The *save* command will add some data to the end of the filename – this helps to keep track of what each saved file is.

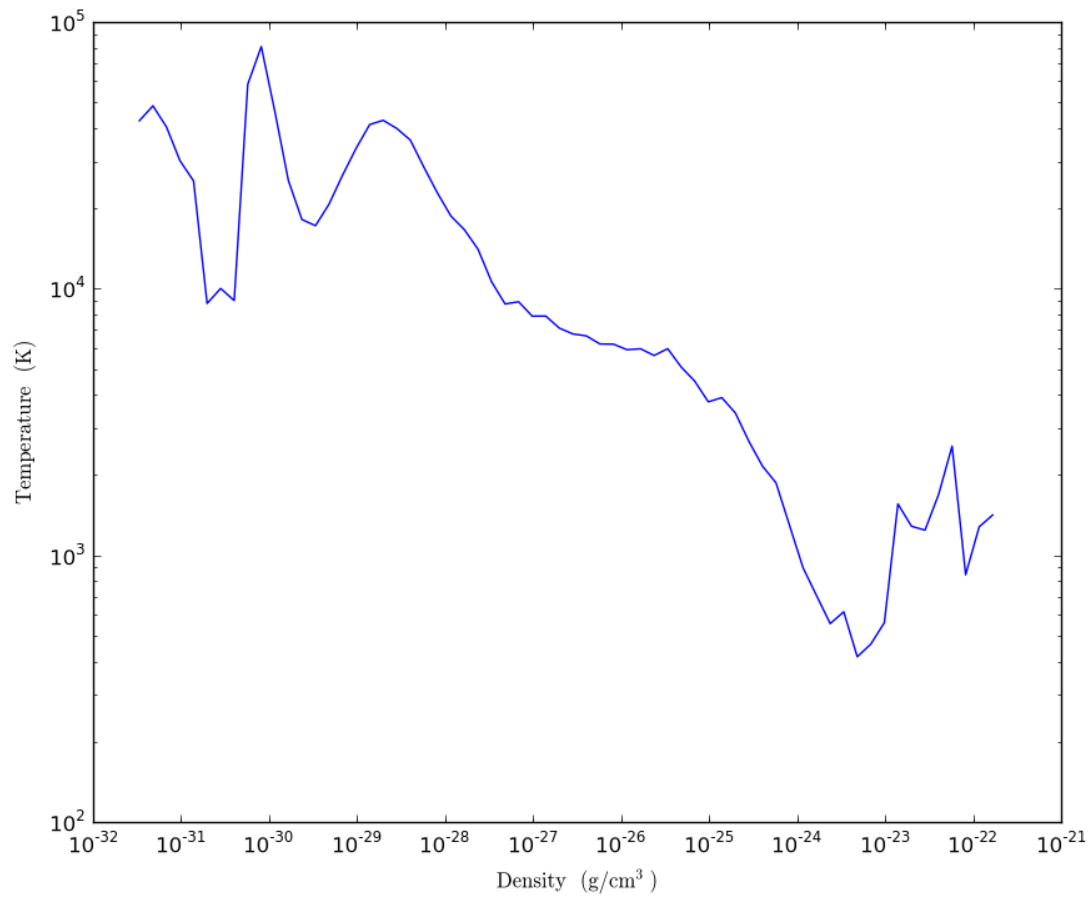
### 4.1.5 A Few More Plots

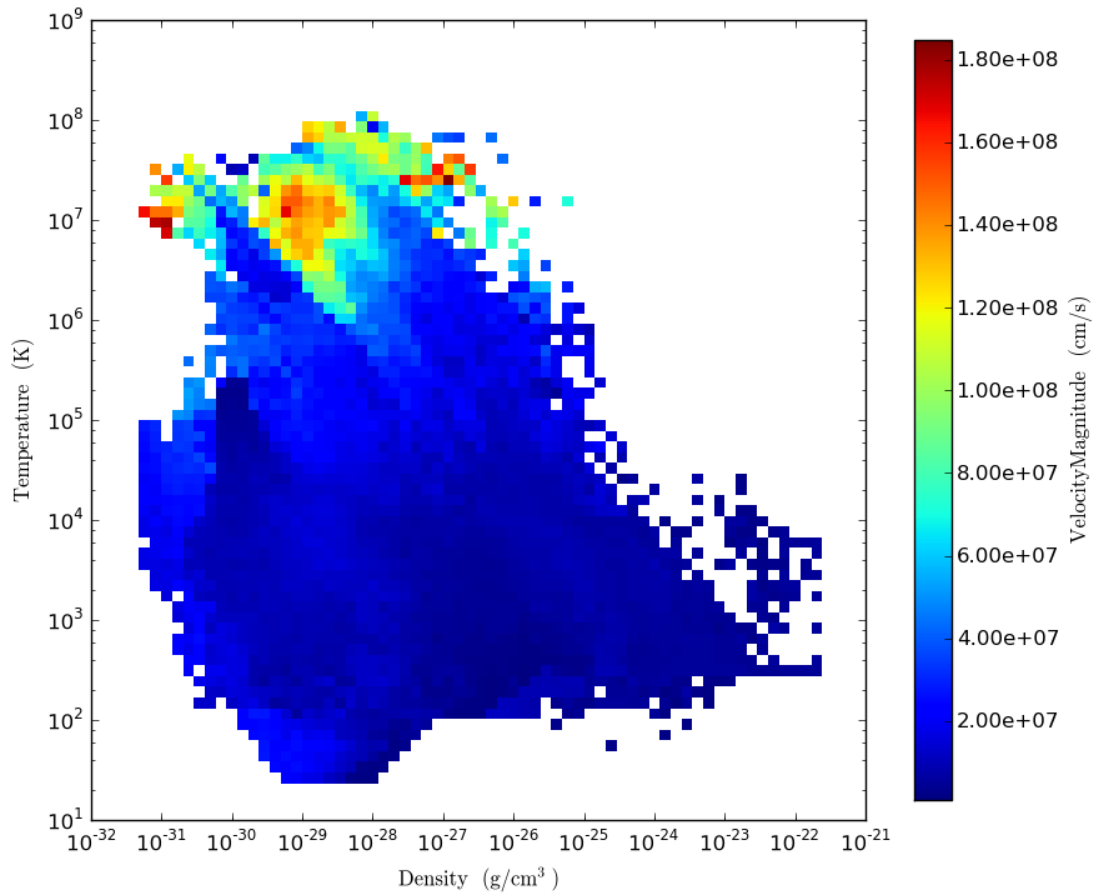
You can also add profiles – radial or otherwise – and phase diagrams very easily.

```
In [8]: pc.add_profile_sphere(100.0, 'kpc', ["Density", "Temperature"])
Out[8]: <yt.raven.PlotTypes.Profile1DPlot instance at 0xada03ec>
```

```
In [9]: pc.add_phase_sphere(100.0, 'kpc', ['Density', 'Temperature',
...:                                       'VelocityMagnitude'])
Out[9]: <yt.raven.PlotTypes.PhasePlot instance at 0xada91ef>
```



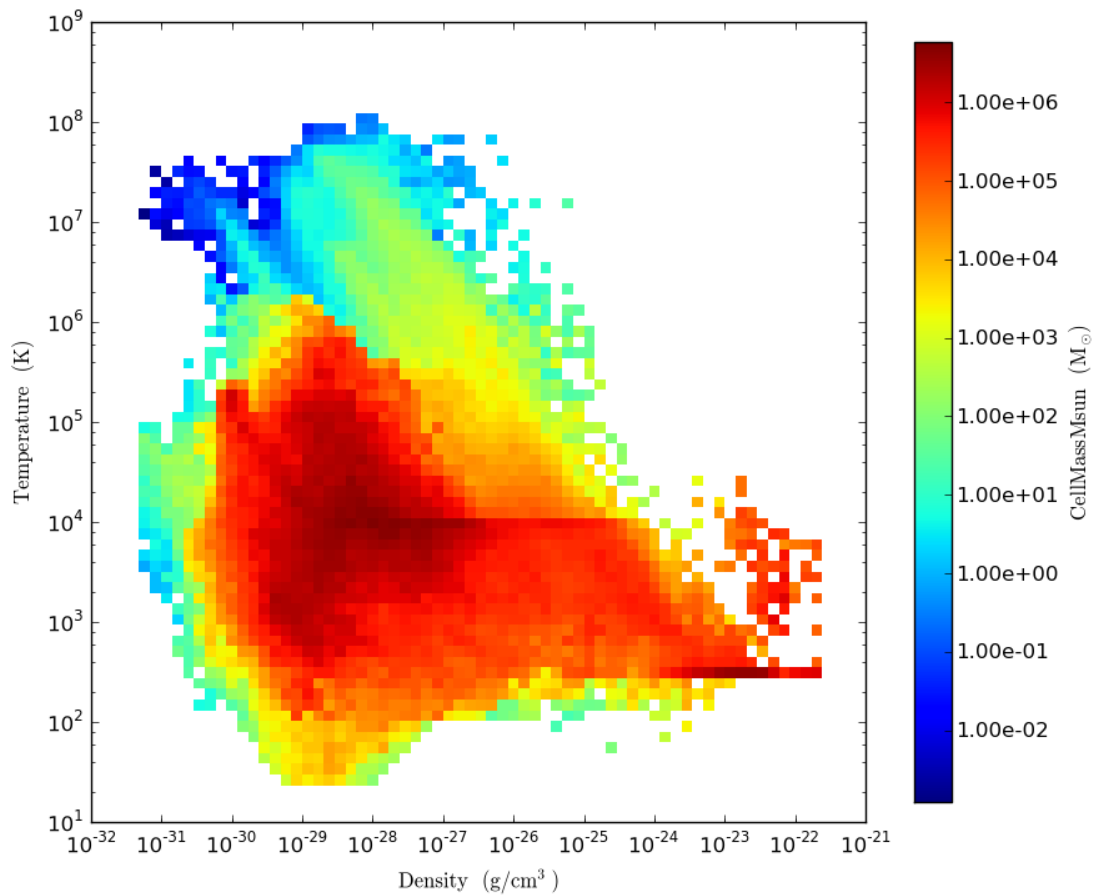




Note that the phase plots default to showing a weighted-average in each bin – weighted by the cell mass in solar masses. If you want to see a distribution of mass, you’ll need to specify you don’t want an average:

```
In [10]: pc.add_phase_sphere(100.0, 'kpc', ['Density', 'Temperature',
...:                                       'CellMassMsun'], weight=None)
```

```
Out[10]: <yt.raven.PlotTypes.PhasePlot instance at 0xada91ef>
```



## 4.2 A Slightly Longer Introduction

This section will contain a short (but a bit longer than *Quick Start Guide*) introduction to analyzing and plotting data with `yt`, using a scripting interface. If you're not familiar with Python, you might be able to pick it up from this section, but you'd probably be better off reading one of the many other sources listed in *Where can I learn more about Python?*)

**Note:** If you know Python, you might enjoy reading *Cookbook*!

### 4.2.1 Writing a Script

The very first step to using `yt` is to open up a text editor, write a little script, and then run it. You can use your favorite text editor (for instance, `vim`) and then save it as something ending in `.py`. At the command line, you can execute this script by calling the name of the python interpreter that you used to install `yt` and then the name of the script:

```
$ python2.6 my_script.py
```

This will load the interpreter, read and run the script `my_script.py` and then terminate regardless of the success or failure of the script.

To have the python interpreter load, run, and then return an interactive prompt, you can execute the script with

```
$ python2.6 -i my_script.py
```

There's a bit more information about invocation of python in *Debugging and Driving YT*.

Okay, so now we know how to launch a script, but what do we put in it? Let's start with one of the most simple things to do. Let's load some data and find the most dense point. Here's a sample little script that loads our data, prints the maximum density, and the location of that maximum density.

We first import `yt` – the very first line in this sample script loads `yt`, brings a bunch of variables, functions and classes into the local namespace, and initializes a few settings.

The next line loads the parameter file into memory, and then we find the maximum density.

```
from yt.mods import *
pf = load("RedshiftOutput0010.dir/RedshiftOutput0010")
value, position = pf.h.find_max("Density")

print "Maximum density: %0.5e at %s" % (value, position)
```

The last line in that script is a format string which prints the value and the position. You can find a number of sample recipes in the *Cookbook*. Let's move on to making a script that makes some plots before terminating.

## 4.2.2 Plots and Plot Types

The next step we might want to take is to visually inspect our data. `yt` has a facility for creating several linked plots – `yt.raven.PlotCollection` handles adding multiple plots that are linked by width and parameter files. We can add a couple slices, along each axis and then zoom in. This is one of the most fundamental idioms in `yt` – during my thesis work, almost all of my scripts started out like this.

```
from yt.mods import *
pf = load("DataDump0020.dir/DataDump0020")
pc = PlotCollection(pf)

pc.add_slice("Density", 0)
pc.add_slice("Temperature", 0)

pc.set_width(1000.0, 'au')

pc.save()
```

This particular script will create a `PlotCollection` centered on the most dense point (unless you feed in a center, it searches for and finds the most dense point) add a `Density` slice, a `Temperature` slice, set the width to 1000.0 AU, and then save the lot of them.

For more complicated examples, be sure to check out the *Cookbook* and the API for `yt.raven.PlotCollection` as well as the `yt.raven` documentation as a whole.

## 4.2.3 Plot Modification

`yt` comes with a number of mechanisms of adding visual and textual information to plots. These include grid boundaries, scale boxes, vector fields, contour fields and text annotations. More documentation is available in *Plot Modification Mechanisms*, with full API documentation in `yt.raven.Callbacks`.

The plot modifications all follow a uniform interface; the concept is that each plot has a base plot, and on top of that a set of *callbacks* that are applied, in order, to modify it and produce a final result. To apply a new modification, the `modify` dictionary of the plot is accessed, and from that the appropriate modification keyword is selected. Each of these accepts a set of arguments.

For example, from start to finish, this command will output a slice through the most dense point in the simulation, taken along the x axis, with the grid boundaries drawn.

```
from yt.mods import *
p = plots.get_slice("my_data0001", "Density", 0)
p.modify["grids"]()
p.save_image("my_data0001_Density")
```

To add on a contour of the field “Temperature”, you can add on another modification:

```
p.modify["contour"]("Temperature")
p.save_image("my_data0001_Density_Temperature")
```

The plots returned by the class-`yt.raven.PlotCollection` methods also respect this interface, which means that you can also do things like:

```
from yt.mods import *
pf = load("my_data0001")
pc = PlotCollection(pf)
for ax in range(3): pc.add_slice("Density", ax).modify["grids"]()
pc.save("my_data0001")
```

Wrapped up into this snippet are the methods for adding slices along all three axes and then instantly applying to them the grid boundary outlines.

A full list of the different possibilities for plot modifications is available in *Plot Modification Mechanisms*.

## 4.2.4 Time Series Movies

**Note:** The *Command Line Tool* can also do time series plots. Here we showcase how to do them from a script so that more modifications can be made.

The process of constructing a time series movie involves, very simply, constructing a set of plots over a set of parameter files. By iterating over a set of data files, or over a set of numbers, a series of plots can be output. These can then be concatenated into a movie to show changes in features and fields over time.

For example, the simplest possible time series movie script would be:

```
from yt.mods import *
for i in range(1000):
    p = plots.get_slice("my_data%04i" % i, "Density", 0)
    p.save_image("my_data%04i" % i)
```

Because we are using the full API here, more complicated visualizations can be built up. For instance, with the addition of

```
p.set_width(10, 'kpc')
p.set_zlim(1e-27, 1e-24)
```

the width of each image will be 10 kpc and the color limits will be set to  $1e-27$  and  $1e-24$ .

### 4.2.5 Even More!

There's quite a bit more that you can do with `yt` from a scripting perspective – not only can you use the modules that come with `yt`, but you can use all of the modules available for Python as a whole. [SciPy](#) is a good starting point, and there are lots of fun subpackages as well as other scientific plotting packages available.

If you find something cool that you find a neat way to apply to Adaptive Mesh Refinement data, you should be sure to email [The Mailing List](#) to tell us about it!

## 4.3 Command Line Tool

`yt` comes with a command-line tool, known as `yt`, that exposes much of the functionality that would normally be accessible through a scripts. This is designed to make the process of making immediate plots much easier. All of the functionality is described in the help strings:

```
$ yt help
```

and then the subcommands all have `help` options as well:

```
$ yt plot --help
```

In order to actually run the command, you'll need to tell it which outputs to operate on. The `yt` command-line tool has three mechanisms for specifying outputs. It will do its best to guess based on the information its provided.

You can specify a base name for a parameter file and then a start and stop number (and optionally a skip parameter):

```
$ yt plot --basename=RedshiftOutput --skip 5 10 50
```

This will run your plot command on `RedshiftOutput` 10 through 50, but only on multiples of five. (And if your output is in a subdirectory, `yt` will check there too, don't worry!)

You can specify a single parameter file:

```
$ yt plot RedshiftOutput0010
```

This will run your plot command on `RedshiftOutput0010`.

Alternatively, you can specify multiple parameter files on the command line:

```
$ yt plot RedshiftOutput0010 RedshiftOutput0020 RedshiftOutput0030
```

This will plot `RedshiftOutput0010`, `RedshiftOutput0020`, and `RedshiftOutput0030`.

### 4.3.1 Simple Statistics

To get information about a given parameter file, including the maximum density, the level information, the smallest cell size and some timing information, use the `stats` command:

```
$ yt stats RedshiftOutput0005
```

0	4	32768
1	34	253496
2	304	525784
-----		
	342	812048

```
z = 0.00000000
t = 6.46750660e+02 = 4.57786981e+17 s = 1.45163299e+10 years
```

Smallest Cell:

```
Width: 7.812e-03 1
Width: 7.812e-03 unitary
Width: 3.906e-02 mpch
Width: 3.906e-02 mpchcm
Width: 6.010e-02 mpc
Width: 7.812e-01 aye
Width: 3.906e+01 kpch
Width: 3.906e+01 kpchcm
Width: 6.010e+01 kpc
Width: 3.906e+04 pch
Width: 3.906e+04 pchcm
Width: 6.010e+04 pc
Width: 8.059e+09 auh
Width: 8.059e+09 auhcm
Width: 1.240e+10 au
Width: 8.659e+11 rsunh
Width: 8.659e+11 rsunhcm
Width: 1.332e+12 rsun
Width: 7.488e+17 miles
Width: 7.488e+17 miles
Width: 1.152e+18 miles
Width: 1.205e+23 cmh
Width: 1.205e+23 cmhcm
Width: 1.854e+23 cm
```

Maximum density: 4.43898e-27 at (0.94921875, 0.80078125, 0.61328125)

## 4.3.2 Plots

The command line tool can make either projections or slices. To make a projection, supply it with the `-p` option:

```
$ yt plot -p RedshiftOutput0005
```

If you don't supply the `-p` option, it will only slice rather than project through the object. Weights can also be supplied for an average along the line of sight. This command defaults to the full width, centered on the most dense point, and outputting along all three axes. The help command has more information:

Create a set of images

Usage:

```
yt plot [ARGS...]
```

Options:

```
-h, --help          show this help message and exit
-w WIDTH, --width=WIDTH
```

```
                                Width in specified units
-u UNIT, --unit=UNIT
                                Desired units
-b BASENAME, --basename=BASENAME
                                Basename of parameter files
-p, --projection               Use a projection rather than a slice
-c CENTER, --center=CENTER
                                Center (-1,-1,-1 for max)
-z ZLIM, --zlim=ZLIM
                                Color limits (min, max)
-a AXIS, --axis=AXIS
                                Axis (4 for all three)
-f FIELD, --field=FIELD
                                Field to color by
-g WEIGHT, --weight=WEIGHT
                                Field to weight projections with
-s SKIP, --skip=SKIP
                                Skip factor for outputs
--colormap=CMAP                Colormap name
-o OUTPUT, --output=OUTPUT
                                Folder in which to place output images
--show-grids                   Show the grid boundaries
```

### 4.3.3 Zoomin Movies

The command line tool also has facilities for outputting a set of frames that zoom in on a central position. This works on a single dataset and can zoom in on projections or slices:

```
$ yt zoomin RedshiftOutput0005
```

However, as with the other commands, you will likely want to specify your own options.

Create a set of zoomin frames

Options:

```
-h, --help                    show this help message and exit
--max-width=MAX_WIDTH
                                Maximum width in code units
--min-width=MIN_WIDTH
                                Minimum width in units of smallest dx (default: 50)
-p, --projection               Use a projection rather than a slice
-a AXIS, --axis=AXIS
                                Axis (4 for all three)
-f FIELD, --field=FIELD
                                Field to color by
-g WEIGHT, --weight=WEIGHT
                                Field to weight projections with
-z ZLIM, --zlim=ZLIM
                                Color limits (min, max)
-n NFRAMES, --nframes=NFRAMES
                                Number of frames to generate
-o OUTPUT, --output=OUTPUT
                                Folder in which to place output images
--colormap=CMAP                Colormap name
--unit-boxes                   Display helpful unit boxes
--dex=DEX                      Number of dex above min to display
```



```
-t TEXT, --text=TEXT
    Textual annotation
```

#### 4.3.4 Halo Profiler

### 4.4 Using and Manipulating Objects and Fields

To generate standard plots, objects rarely need to be directly constructed. However, for detailed data inspection as well as hand-crafted derived data, objects can be exceptionally useful and even necessary.

#### 4.4.1 Accessing Fields in Objects

yt utilizes load-on-demand objects to represent physical regions in space. (See *Object Methodology*.) Data objects in yt all respect the following protocol for accessing data:

```
my_object["Density"]
```

where "Density" can be any field name. The full list of objects is available in *Available Objects*, and information about how to create an object can be found in *Creating 3D Datatypes*. The field is returned as a single, flattened array without spatial information. The best mechanism for manipulating spatial data is the `CoveringGridBase` object.

The full list of fields that are available can be found as a property of the Hierarchy or Static Output object that you wish to access. This property is calculated every time the object is instantiated. The full list of fields that have been identified in the output file, which need no processing (besides unit conversion) are in the property `field_list` and the full list of potentially-accessible derived fields (see *Derived Fields and Derived Quantities*) is available in the property `derived_field_list`. You can see these by examining the two properties:

```
pf = load("my_data")
print pf.h.field_list
print pf.h.derived_field_list
```

When a field is added, it is added to a container that hangs off of the parameter file, as well. All of the field creation options (*Field Options*) are accessible through this object:

```
pf = load("my_data")
print pf.h.field_info["Pressure"].units
```

This is a fast way to examine the units of a given field, and additionally you can use `yt.lagos.DerivedField.get_source()` to get the source code:

```
field = pf.h.field_info["Pressure"]
print field.get_source()
```

#### 4.4.2 Available Objects

Objects are instantiated by direct access of a hierarchy. Each of the objects that can be generated by a hierarchy are in fact fully-fledged data objects respecting the standard protocol for interaction.

The following objects are available, all of which hang off of the hierarchy object. To access them, you would do something like this (as for a `region`):

```
from yt.mods import *
pf = load("RedshiftOutput0005")
reg = pf.h.region([0.5, 0.5, 0.5], [0.0, 0.0, 0.0], [1.0, 1.0, 1.0])
```

**class covering\_grid(self, level, left\_edge, right\_edge, dims, fields=None, pf=None, num\_ghosts=None):**  
(This is a proxy for `yt.lagos.AMRCoveringGridBase`.) The data object returned will consider grids up to *level* in generating fixed resolution data between *left\_edge* and *right\_edge* that is *dims* (3-values) on a side.

**class cutting(self, normal, center, fields=None, node\_name=None, \*\*field\_parameters):**  
(This is a proxy for `yt.lagos.AMRCuttingPlaneBase`.) The Cutting Plane slices at an oblique angle, where we use the *normal* vector and the *center* to define the viewing plane. The ‘up’ direction is guessed at automatically.

**class disk(self, center, normal, radius, height, fields=None, pf=None, \*\*field\_parameters):**  
(This is a proxy for `yt.lagos.AMRCylinderBase`.) By providing a *center*, a *normal*, a *radius* and a *height* we can define a cylinder of any proportion. Only cells whose centers are within the cylinder will be selected.

**class extracted\_region(self, base\_region, indices, force\_refresh=True, \*\*field\_parameters):**  
(This is a proxy for `yt.lagos.ExtractedRegionBase`.) Returns an instance of `AMR3DData`, or prepares one. Usually only used as a base class. Note that *center* is supplied, but only used for fields and quantities that require it.

**class grid(self, id, filename=None, hierarchy=None):**  
(This is a proxy for `yt.lagos.EnzoGridBase`.) Returns an instance of `EnzoGrid` with *id*, associated with *filename* and *hierarchy*.

**class grid\_collection(self, center, grid\_list, fields=None, pf=None, \*\*field\_parameters):**  
(This is a proxy for `yt.lagos.AMRGridCollectionBase`.) By selecting an arbitrary *grid\_list*, we can act on those grids. Child cells are not returned.

**class ortho\_ray(self, axis, coords, fields=None, pf=None, \*\*field\_parameters):**  
(This is a proxy for `yt.lagos.AMROrthoRayBase`.) Dimensionality is reduced to one, and an ordered list of points at an (x,y) tuple along *axis* are available.

**class periodic\_region(self, center, left\_edge, right\_edge, fields=None, pf=None, \*\*field\_parameters):**  
(This is a proxy for `yt.lagos.AMRPeriodicRegionBase`.) We create an object with a set of three *left\_edge* coordinates, three *right\_edge* coordinates, and a *center* that need not be the center.

**class periodic\_region\_strict(self, center, left\_edge, right\_edge, fields=None, pf=None, \*\*field\_parameters):**  
(This is a proxy for `yt.lagos.AMRPeriodicRegionStrictBase`.) We create an object with a set of three *left\_edge* coordinates, three *right\_edge* coordinates, and a *center* that need not be the center.

**class proj(self, axis, field, weight\_field=None, max\_level=None, center=None, pf=None, source=None):**  
(This is a proxy for `yt.lagos.AMRProjBase`.) `AMRProj` is a projection of a *field* along an *axis*. The field can have an associated *weight\_field*, in which case the values are multiplied by a weight before being summed, and then divided by the sum of that weight.

**class ray(self, start\_point, end\_point, fields=None, pf=None, \*\*field\_parameters):**  
(This is a proxy for `yt.lagos.AMRRayBase`.) We accept a start point and an end point and then get all the data between those two.

**class region(self, center, left\_edge, right\_edge, fields=None, pf=None, \*\*field\_parameters):**  
(This is a proxy for `yt.lagos.AMRRegionBase`.) We create an object with a set of three *left\_edge* coordinates, three *right\_edge* coordinates, and a *center* that need not be the center.

**class region\_strict(self, center, left\_edge, right\_edge, fields=None, pf=None, \*\*field\_parameters):**  
(This is a proxy for `yt.lagos.AMRRegionStrictBase`.) We create an object with a set of three *left\_edge* coordinates, three *right\_edge* coordinates, and a *center* that need not be the center.

```
class slice(self, axis, coord, fields=None, center=None, pf=None, node_name=False, **field_pa
(This is a proxy for yt.lagos.AMRSliceBase.) Slice along axis How do I specify an axis?, at the coordinate coord. Optionally supply fields.
```

```
class smoothed_covering_grid(self, *args, **field_parameters):()
(This is a proxy for yt.lagos.AMRSmoothedCoveringGridBase.) The data object returned will consider grids up to level in generating fixed resolution data between left_edge and right_edge that is dims (3-values) on a side.
```

```
class sphere(self, center, radius, fields=None, pf=None, **field_parameters):()
(This is a proxy for yt.lagos.AMRSphereBase.) The most famous of all the data objects, we define it via a center and a radius.
```

### 4.4.3 Storing and Loading Objects

Often, when operating interactively or via the scripting interface, it is convenient to save an object or multiple objects out to disk and then restart the calculation later. Personally, I found this most useful when dealing with identification of clumps and contours (see [Cookbook](#) for a recipe on how to find clumps and the API documentation for both `ContourFinder` and `Clump`) where the identification step can be quite time-consuming, but the analysis may be relatively fast.

Typically, the save and load operations are used on 3D data objects. `yt` has a separate set of serialization operations for 2D objects such as projections. `yt` will save out 3D objects to disk under the presupposition that the construction of the objects is the difficult part, rather than the generation of the data – this means that you can save out an object as a description of how to recreate it in space, but not the actual data arrays affiliated with that object. The information that is saved includes the parameter file off of which the object “hangs.” It is this piece of information that is the most difficult; the object, when reloaded, must be able to reconstruct a parameter file from whatever limited information it has in the save file.

To do this, `yt` is able to identify parameter files based on a “hash” generated from the base file name, the “Current-TimeIdentifier”, and the simulation time. These three characteristics should never be changed outside of a simulation, they are independent of the file location on disk, and in conjunction they should be uniquely identifying. (This process is all done in `fido` via `ParameterFileStore`.)

To save an object, you can either save it in the `.yt` file affiliated with the hierarchy (*What are all these .yt files?*) or as a standalone file. For instance, using `save_object()` we can save a sphere.

```
from yt.mods import *
pf = load("my_data")
sp = pf.h.sphere([0.5, 0.5, 0.5], 10.0/pf['kpc'])

pf.h.save_object(sp, "sphere_to_analyze_later")
```

In a later session, we can load it using `save_object()`:

```
from yt.mods import *

pf = load("my_data")
sphere_to_analyze = pf.h.load_object("sphere_to_analyze_later")
```

Additionally, if we want to store the object independent of the `.yt` file, we can save the object directly:

```
from yt.mods import *

pf = load("my_data")
sp = pf.h.sphere([0.5, 0.5, 0.5], 10.0/pf['kpc'])
```

```
sp.save_object("my_sphere", "my_storage_file.cpk1")
```

This will store the object as `my_sphere` in the file `my_storage_file.cpk1`, which will be created or accessed using the standard python module `shelve`. Note that if a filename is not supplied, it will be saved via the hierarchy, as above.

To re-load an object saved this way, you can use the `shelve` module directly:

```
from yt.mods import *
import shelve

obj_file = shelve.open("my_storage_file.cpk1")
pf, obj = obj_file["my_sphere"]
```

Note here that this behaves slightly differently than above – we do not need to load the parameter file ourselves, as the load process actually does that for us! Additionally, we can store multiple objects in a single `shelve` file, so we have to call the sphere by name.

**Note:** It's also possible to use the standard `cPickle` module for loading and storing objects – so in theory you could even save a list of objects!

This method works for clumps, as well, and the entire clump hierarchy will be stored and restored upon load.

## 4.5 Examining and Manipulating Particles

`yt` has support for reading and manipulating particles. You can access the particles as you would any other data field; additionally, derived fields that operate on particles can be added as would any other derived field, as long as the parameter *particle\_type* is set to `True` in the call to `add_field()`. However, with that, there are a few caveats. Particle support in `yt` is not by any means an afterthought, but it was developed relatively late in comparison to baryon and field-based analysis, and is not as mature.

**Note:** If you are having trouble with particles, email the mailing list!

### 4.5.1 Using Particles

Many particle operations can be conducted obliquely, which will serve to reduce memory usage as well as handle any problems that might arise from spatial selection of particles.

For instance, `Halo` objects have a number of operations that can transparently calculate center of mass of particles, bulk velocity, and so on. Use those instead of obtaining the fields directly. Furthermore, any of the spatially-addressable objects described in *Using and Manipulating Objects and Fields* will automatically select particles based on the region of space they describe, and the quantities (*Derived Quantities*) in those objects will operate on particle fields.

(For information on halo finding, see *Halo finding* and *Halo mass info*.)

**Warning:** If you use the built-in methods of interacting with particles, you should be well off. Otherwise, there are caveats!

### 4.5.2 Selection By Type

Unfortunately, Enzo’s mechanism for storing particle type is inconsistent. The parameter `ParticleTypeInFile` controls whether or not the field `particle_type` is written to disk; if it is set to 1, the field will be written, but the default is 0 where the field is not written. Without the field `particle_type` the discriminator between particle types is exclusively based on the field `creation_time`. Particles with `creation_time` greater than 0.0 are star particles and those with `creation_time` equal to zero are dark matter particles.

For simulations only including dark matter particles, this is not important, as all of the particles will be of the same type. However, selection of – for instance – star particles in other simulations will require some care, and you will need to do it differently depending on the value of `ParticleTypeInFile`.

#### Selecting Particles By Creation Time

To select particles based on creation time, you must first create an index array. Python (and NumPy) allow indexing based on boolean values, so we will do that. Here is an example of selecting all star particles in the domain.

```
from yt.mods import *
pf = load("galaxy1200.dir/galaxy1200")
dd = pf.h.all_data()

star_particles = dd["creation_time"] > 0.0
print dd["ParticleMassMsun"][star_particles].max()
print dd["ParticleMassMsun"][star_particles].min()
print "Number of star particles", star_particles.sum()
```

#### Selecting Particles By Particle Type

In Enzo, star particles are type 2. So we will select using the boolean array (as in *Selecting Particles By Creation Time*) to select only the star particles.

```
from yt.mods import *
pf = load("galaxy1200.dir/galaxy1200")
dd = pf.h.all_data()

star_particles = dd["particle_type"] == 2
print dd["ParticleMassMsun"][star_particles].max()
print dd["ParticleMassMsun"][star_particles].min()
print "Number of star particles", star_particles.sum()
```

### 4.5.3 Memory

Unfortunately, as of right now, particle loading via spatially-selected objects can be memory intensive. The process that `yt` goes through to load particles into memory in a 3D data object is to separate the grids into two classes:

- Fully-contained grids
- Partially-contained grids

For the grids in the former category, the full set of particles residing in those grids are loaded. The ones in the second require that a `FakeGridForParticles` be created so that the particles residing in the region (as determined by their values of `particle_position_x`, `particle_position_y` and `particle_position_z`, which must be loaded from disk) can be selected and cut from the full set of particles. This requires that the full position information for the particles be loaded, which increases overall memory usage.

### 4.5.4 The Future

The next version of `yt` will have a completely rewritten particle infrastructure. This version is currently in the testing phase, but has shown to reduce memory overhead substantially as well as increase speed by a factor of a few. Both spatial selection (selection within an object) and selection by type are extremely promising.

## 4.6 Creating Derived Fields

One of the more powerful means of extending `yt` is through the usage of derived fields. These are fields that describe a value at each cell in a simulation.

### 4.6.1 Defining a New Field

So once a new field has been conceived of, the best way to create it is to construct a function that performs an array operation – operating on a collection of data, neutral to its size, shape, and type. (All fields should be provided as 64-bit floats.)

A simple example of this is the pressure field, which demonstrates the ease of this approach.

```
def _Pressure(field, data):
    return (data.pf["Gamma"] - 1.0) * \
           data["Density"] * data["ThermalEnergy"]
```

Note that we do a couple different things here. We access the “Gamma” parameter from the parameter file, we access the “Density” field and we access the “ThermalEnergy” field. “ThermalEnergy” is, in fact, another derived field! (“ThermalEnergy” deals with the distinction in storage of energy between dual energy formalism and non-DEF.) We don’t do any loops, we don’t do any type-checking, we can simply multiply the three items together.

Once we’ve defined our function, we need to notify `yt` that the field is available. The `add_field()` function is the means of doing this; it has a number of fairly specific parameters that can be passed in, but here we’ll only look at the most basic ones needed for a simple scalar baryon field.

```
add_field("Pressure", function=_Pressure, units=r"\rm{dyne}/\rm{cm}^{\rm{2}}")
```

We feed it the name of the field, the name of the function, and the units. Note that the units parameter is a “raw” string, with some LaTeX-style formatting – Matplotlib actually has a `MathText` rendering engine, so if you include LaTeX it will be rendered appropriately.

We suggest that you name the function that creates a derived field with the intended field name prefixed by a single underscore, as in the `_Pressure` example above.

Note one last thing about this definition; we do not do unit conversion. All of the fields fed into the field are pre-supposed to be in CGS. If the field does not need any constants applied after that, you are done. If it does, you should define a second function that applies the proper multiple in order to return the desired units and use the argument `convert_function` to `add_field` to point to it.

If you find yourself using the same custom-defined fields over and over, you should put them in your plugins file as described in *The Plugin File*.

## 4.6.2 Some More Complicated Examples

But what if we want to do some more fancy stuff? Here's an example of getting parameters from the data object and using those to define the field; specifically, here we obtain the `center` and `height_vector` parameters and use those to define an angle of declination of a point with respect to a disk.

```
def _DiskAngle(field, data):
    # We make both r_vec and h_vec into unit vectors
    center = data.get_field_parameter("center")
    r_vec = na.array([data["x"] - center[0],
                     data["y"] - center[1],
                     data["z"] - center[2]])
    r_vec = r_vec/na.sqrt((r_vec**2.0).sum(axis=0))
    h_vec = na.array(data.get_field_parameter("height_vector"))
    dp = r_vec[0,:] * h_vec[0] \
        + r_vec[1,:] * h_vec[1] \
        + r_vec[2,:] * h_vec[2]
    return na.arccos(dp)
add_field("DiskAngle", take_log=False,
          validators=[ValidateParameter("height_vector"),
                     ValidateParameter("center")],
          display_field=False)
```

Note that we have added a few parameters below the main function; we specify that we do not wish to display this field as logged, that we require both `height_vector` and `center` to be present in a given data object we wish to calculate this for, and we say that it should not be displayed in a drop-down box of fields to display. This is done through the parameter `validators`, which accepts a list of `FieldValidator` objects. These objects define the way in which the field is generated, and when it is able to be created. In this case, we mandate that parameters `center` and `height_vector` are set before creating the field. These are set via `set_field_parameter()`, which can be called on any object that has fields.

We can also define vector fields.

```
def _SpecificAngularMomentum(field, data):
    if data.has_field_parameter("bulk_velocity"):
        bv = data.get_field_parameter("bulk_velocity")
    else: bv = na.zeros(3, dtype='float64')
    xv = data["x-velocity"] - bv[0]
    yv = data["y-velocity"] - bv[1]
    zv = data["z-velocity"] - bv[2]
    center = data.get_field_parameter('center')
    coords = na.array([data['x'], data['y'], data['z']], dtype='float64')
    new_shape = tuple([3] + [1]*(len(coords.shape)-1))
    r_vec = coords - na.reshape(center, new_shape)
    v_vec = na.array([xv, yv, zv], dtype='float64')
    return na.cross(r_vec, v_vec, axis=0)
def _convertSpecificAngularMomentum(data):
    return data.convert("cm")
add_field("SpecificAngularMomentum",
          convert_function=_convertSpecificAngularMomentum, vector_field=True,
          units=r"\rm{cm}^2/\rm{s}", validators=[ValidateParameter('center')])
```

Here we define the `SpecificAngularMomentum` field, optionally taking a `bulk_velocity`, and returning a vector field that needs conversion by the function `_convertSpecificAngularMomentum`.

### 4.6.3 Field Options

The arguments to `add_field()` are passed on to the constructor of `DerivedField`. `add_field()` takes care of finding the arguments *function* and *convert\_function* if it can, however. There are a number of options available, but the only mandatory ones are *name* and possibly *function*.

**name** This is the name of the field – how you refer to it. For instance, `Pressure` or `H2I_Fraction`.

**function** This is a function handle that defines the field

**convert\_function** This is the function that converts the field to CGS. All inputs to this function are mandated to already *be* in CGS.

**units** This is a mathtext (LaTeX-like) string that describes the units.

**projected\_units** This is a mathtext (LaTeX-like) string that describes the units if the field has been projected without a weighting.

**display\_name** This is a name used in the plots

**take\_log** This is *True* or *False* and describes whether the field should be logged when plotted.

**particle\_type** Is this field a *particle* field?

**validators** (*Advanced*) This is a list of `FieldValidator` objects, for instance to mandate spatial data.

**vector\_field** (*Advanced*) Is this field more than one value per cell?

**display\_field** (*Advanced*) Should this field appear in the dropdown box in Reason?

**not\_in\_all** (*Advanced*) If this is *True*, the field may not be in all the grids.

**projection\_conversion** (*Advanced*) Which unit should we multiply by in a projection?

### 4.6.4 How Do Units Work?

Everything is done under the assumption that all of the native Enzo fields that yt knows about are converted to cgs before being handed to any processing routines.

### 4.6.5 Which Enzo Fields Does yt Know About?

- Density
- Temperature
- Gas Energy
- Total Energy
- [xyz]-velocity
- Species fields: HI, HII, Electron, HeI, HeII, HeIII, HM, H2I, H2II, DI, DII, HDI
- Particle mass, velocity,

## 4.7 Parallel Computation With YT

YT has been instrumented with the ability to compute many – most, even – quantities in parallel. This utilizes the package `mpi4py` to parallelize using the Message Passing Interface, typically installed on clusters.



### 4.7.1 Capabilities

Currently, YT is able to perform the following actions in parallel:

- Projections
- Slices
- Cutting planes (oblique slices)
- Derived Quantities (total mass, angular momentum, etc)
- 1-, 2- and 3-D profiles
- Halo finding

This list covers just about every action YT can take! Additionally, almost all scripts will benefit from parallelization without any modification. The goal of Parallel-YT has been to retain API compatibility and abstract all parallelism.

### 4.7.2 Setting Up Parallel YT

To run scripts in parallel, you must first install `mpi4py`. Instructions for doing so are provided on the [MPI4Py website](#). Once that has been accomplished, you're all done! You just need to launch your scripts with `mpirun` and signal to YT that you want to run them in parallel.

For instance, the following script, which we'll save as `my_script.py`:

```
from yt.mods import *
pf = load("RD0035/RedshiftOutput0035")
v, c = pf.h.find_max("Density")
print v, c
pc = PlotCollection(pf, center = [0.5, 0.5, 0.5])
pc.add_projection("Density", 0)
pc.save()
```

Will execute the finding of the maximum density and the projection in parallel if launched in parallel. To do so, at the command line you would execute

```
$ mpirun -np 16 python2.6 my_script.py --parallel
```

if you wanted it to run in parallel. If you run into problems, then you can use *Remote and Disconnected Debugging* to examine what went wrong.

**Warning:** If you manually interact with the filesystem, not through YT, you will have to ensure that you only execute your functions on the root processor. You can do this with the function `:func:only_on_root`.

It's important to note that all of the processes listed in *capabilities* work – and no additional work is necessary to parallelize those processes. Furthermore, the `yt` command itself recognizes the `--parallel` option, so those commands will work in parallel as well.

### 4.7.3 Types of Parallelism

In order to divide up the work, YT will attempt to send different tasks to different processors. However, to minimize inter-process communication, YT will decompose the information in different ways based on the task.

## Spatial Decomposition

During this process, the hierarchy will be decomposed along either all three axes or along an image plane, if the process is that of projection. This type of parallelism is overall less efficient than grid-based parallelism, but it has been shown to obtain good results overall.

## Grid Decomposition

The alternative to spatial decomposition is a simple round-robin of the grids. This process allows YT to pool data access to a given Enzo data file, which ultimately results in faster read times and better parallelism.

## 4.8 How to Make Plots

Through the plotting interface, you can have `yt` automatically generate many of the analysis objects available to you!

The primary plotting interface is through a `PlotCollection` instantiated with a given parameter file and (optionally) a center. See [Making Plots](#) for a brief example of how to generate a `PlotCollection`.

### 4.8.1 Two-Dimensional Images

Whenever a two-dimensional image is created, the plotting object first obtains the necessary data at the *highest resolution*. Every time an image is requested of it – for instance, when the width or field is changed – this high-resolution data is then pixelized and placed in a buffer of fixed size.

Slices are axially-aligned images of data selected at a fixed point on an axis; these are the fastest type of two-dimensional image, as only the correct coordinate data is read from disk and then plotted.

Cutting planes are oblique slices, aligned with a given normal vector. These can be used for face-on images of disks and other objects, as well as a rotational slices. They work just like slices in other ways, but they tend to be a bit slower.

Projections are closer in style to profiles than slices. They can exist either as a summation of the data along every possible ray through the simulation, or an average value along every possible ray. If a `weight_field` is provided, then the data returned is an average; typically you will want to weight with `Density`. If you do not supply a `weight_field` then the returned data is a column sum. These fields are stored in between invocations – this allows for speedier access to a relatively slow process!

### 4.8.2 Profiles and Phase Plots

Profiles and phase plots provide identical API to the generation of profiles themselves, but with a couple convenience interfaces. You can have the plot collection generate a sphere automatically for either one:

```
pc.add_phase_sphere(100.0, 'au', ["Density", "Temperature", "CellMassMsun"],
                    weight = None)
```

This will generate a sphere, a phase plot, and then return to you the plot object.

### 4.8.3 Interactive Plotting

Thanks to the pylab interface in Matplotlib, we have an interactive plot collection available for usage within IPython. Instead of `PlotCollection`, use `PlotCollectionInteractive` – this will generate automatically updating GUI windows with the plots inside them.

### 4.8.4 Callbacks

Callbacks are means of adding things on top of existing plots – like vectors, overplotted lines, and so on and so forth. They have to be added to the plot objects themselves, rather than the `PlotCollection`. You can add them like so:

```
p = pc.add_slice("Density", 0)
p.modify["grids"]()
```

Each Callback has to be instantiated, and then added. You can also access the plot objects inside the `PlotCollection` directly:

```
pc.add_slice("Density", 0)
pc.plots[-1].modify["grids"]()
```

Note that if you are plotting interactively, the `PlotCollection` will need to have `redraw` called on it.

For more information about Callbacks, see the [API reference](#).



---

# COOKBOOK

yt scripts can be a bit intimidating, and at times a bit obtuse. But there's a lot you can do, and this section of the manual will assist with figuring out how to do some fairly common tasks – which can lead to combining these, with other Python code, into more complicated and advanced tasks.

**Note:** All of these scripts are located in the mercurial repository at <http://hg.enzotools.org/cookbook/>

## 5.1 Simple slice

This is a simple recipe to show how to open a dataset and then plot a slice through it, centered at its most dense point.

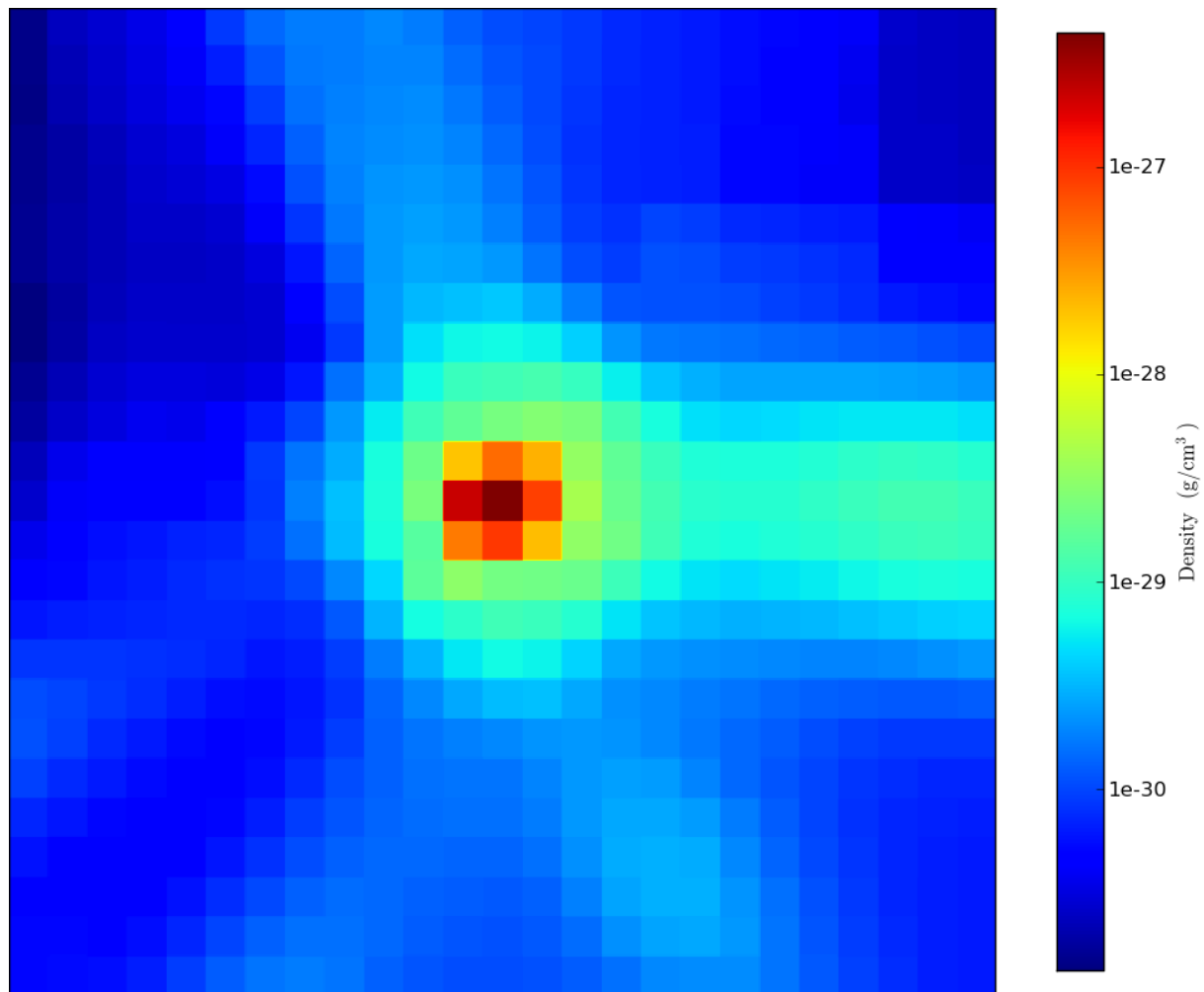
The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/simple\\_slice.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/simple_slice.py).

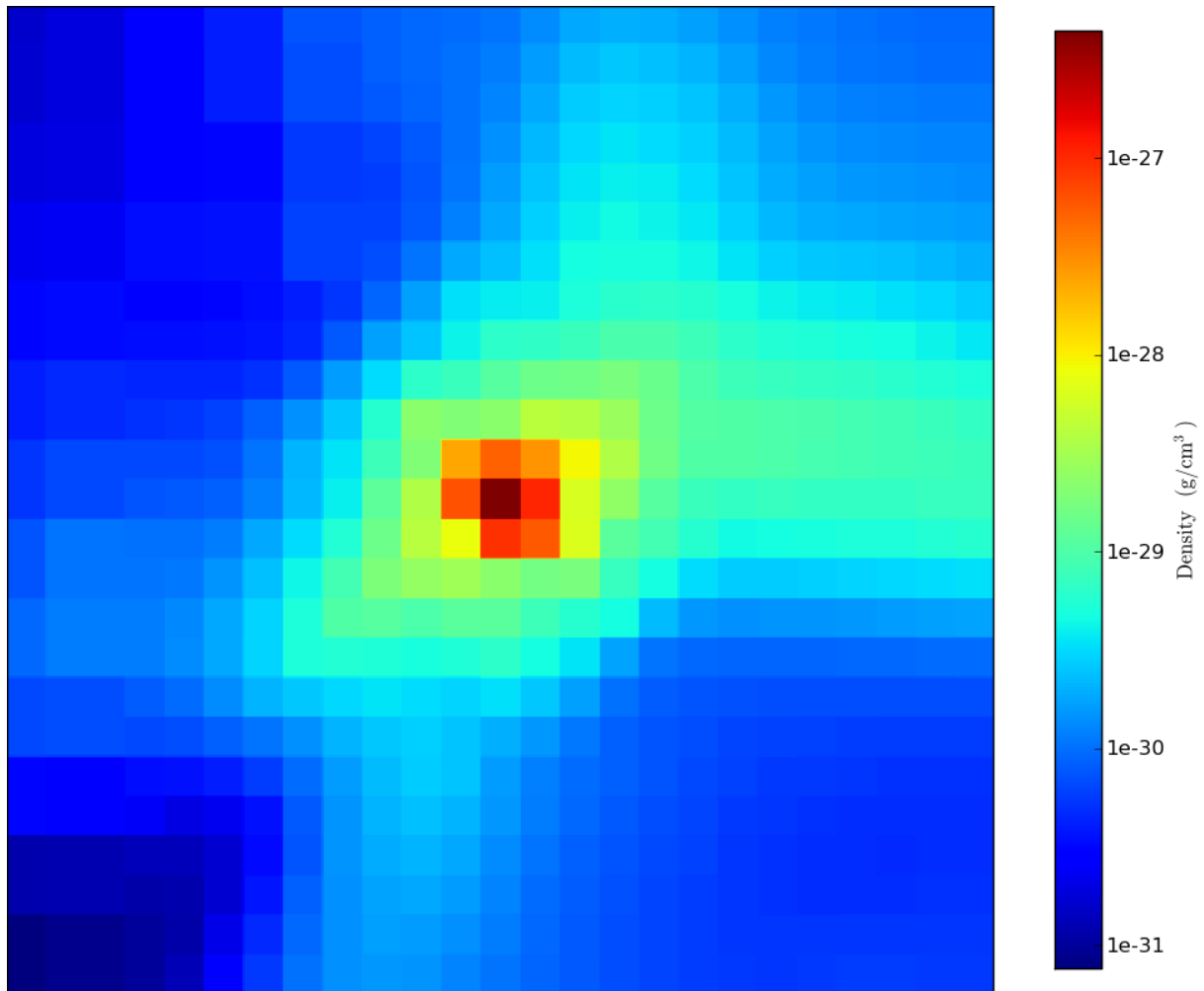
```
from yt.mods import * # set up our namespace

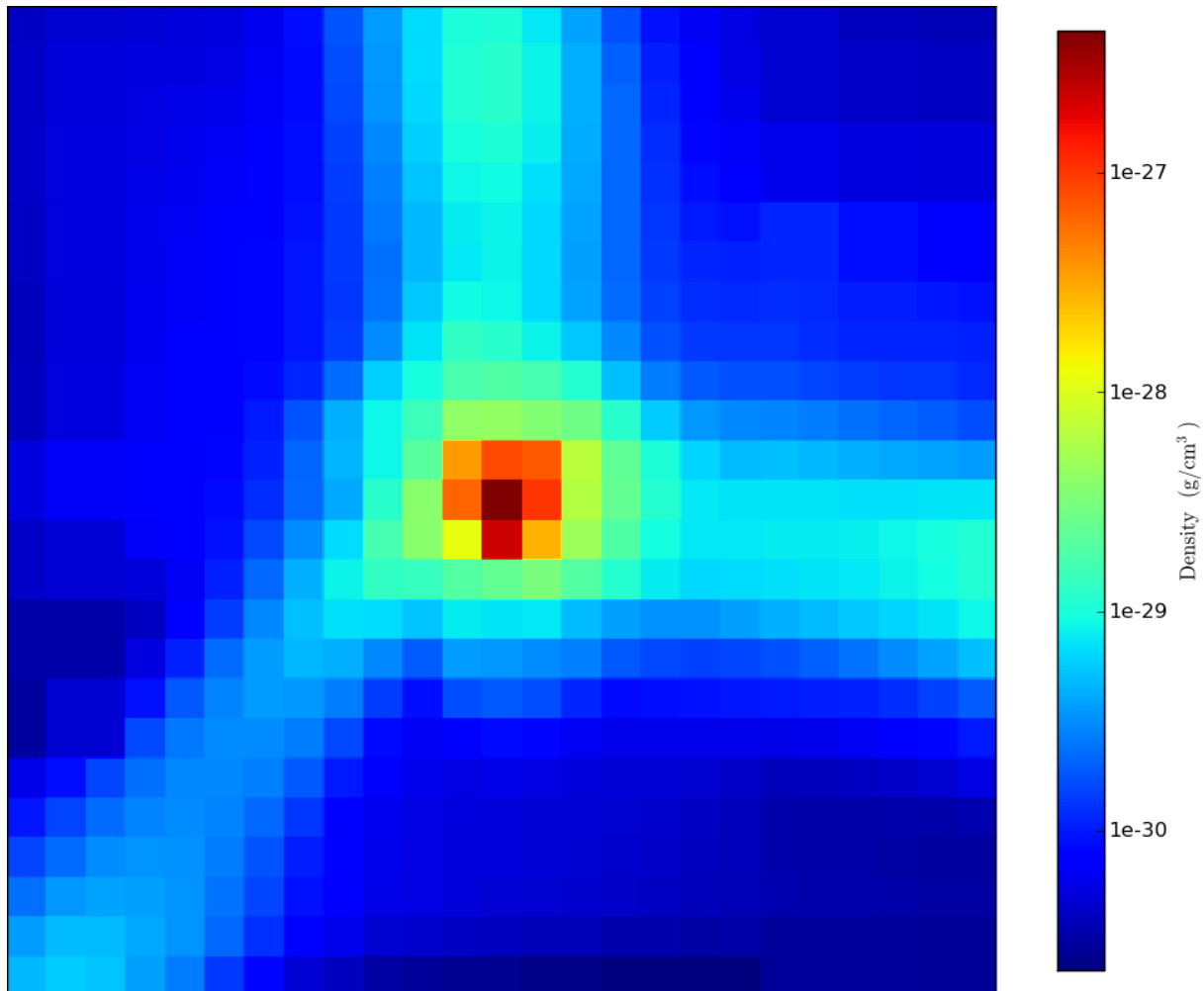
fn = "RedshiftOutput0005" # parameter file to load

pf = load(fn) # load data
pc = PlotCollection(pf) # defaults to center at most dense point
pc.add_slice("Density", 0) # 0 = x-axis
pc.add_slice("Density", 1) # 1 = y-axis
pc.add_slice("Density", 2) # 2 = z-axis
pc.set_width(1.5, 'mpc') # change width of all plots in pc
pc.save(fn) # save all plots
```

## Sample Output







## 5.2 Simple projection

This is a simple recipe to show how to open a dataset and then take a weighted-average projection through it, centered at its most dense point.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/simple\\_projection.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/simple_projection.py).

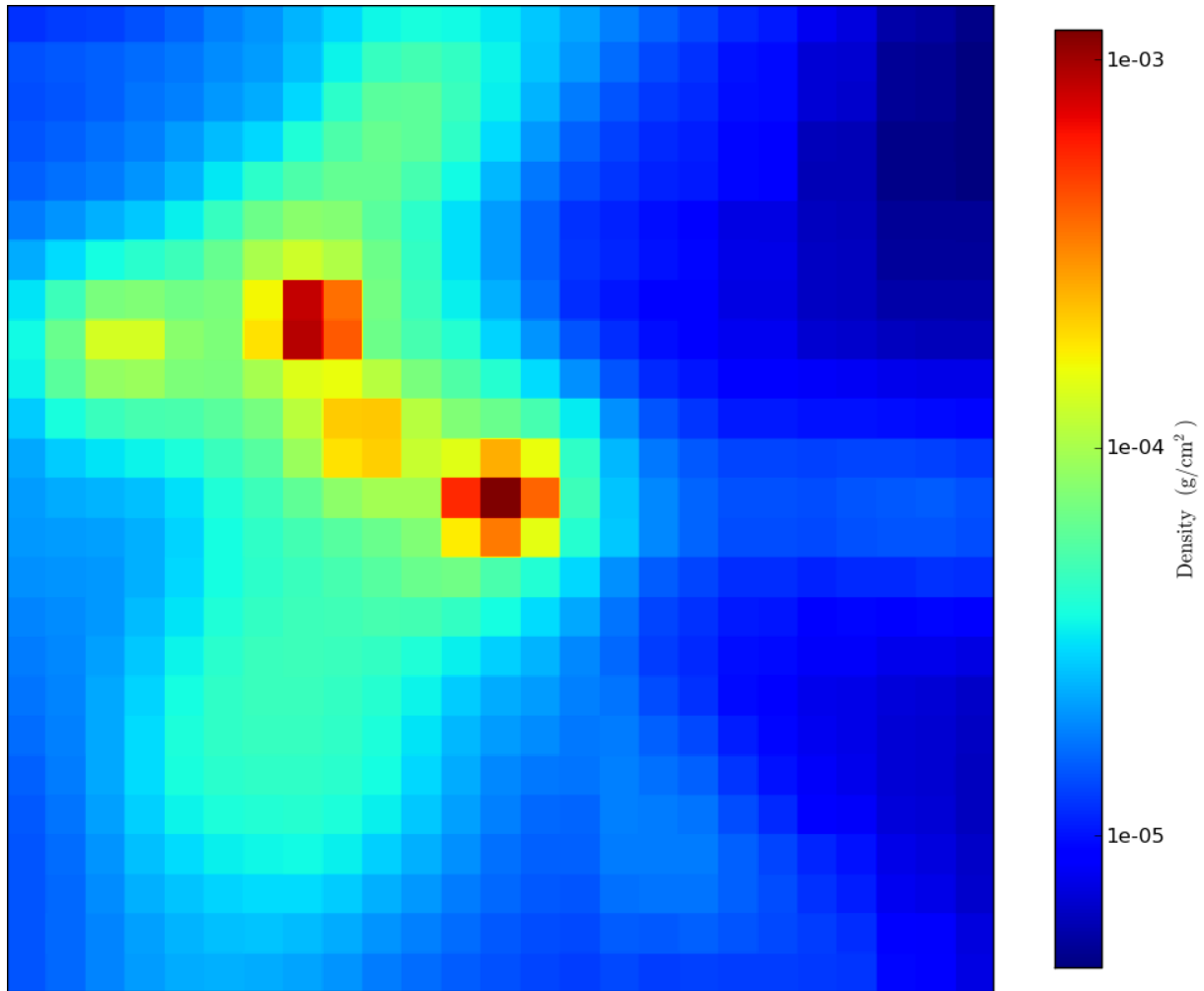
```
from yt.mods import * # set up our namespace

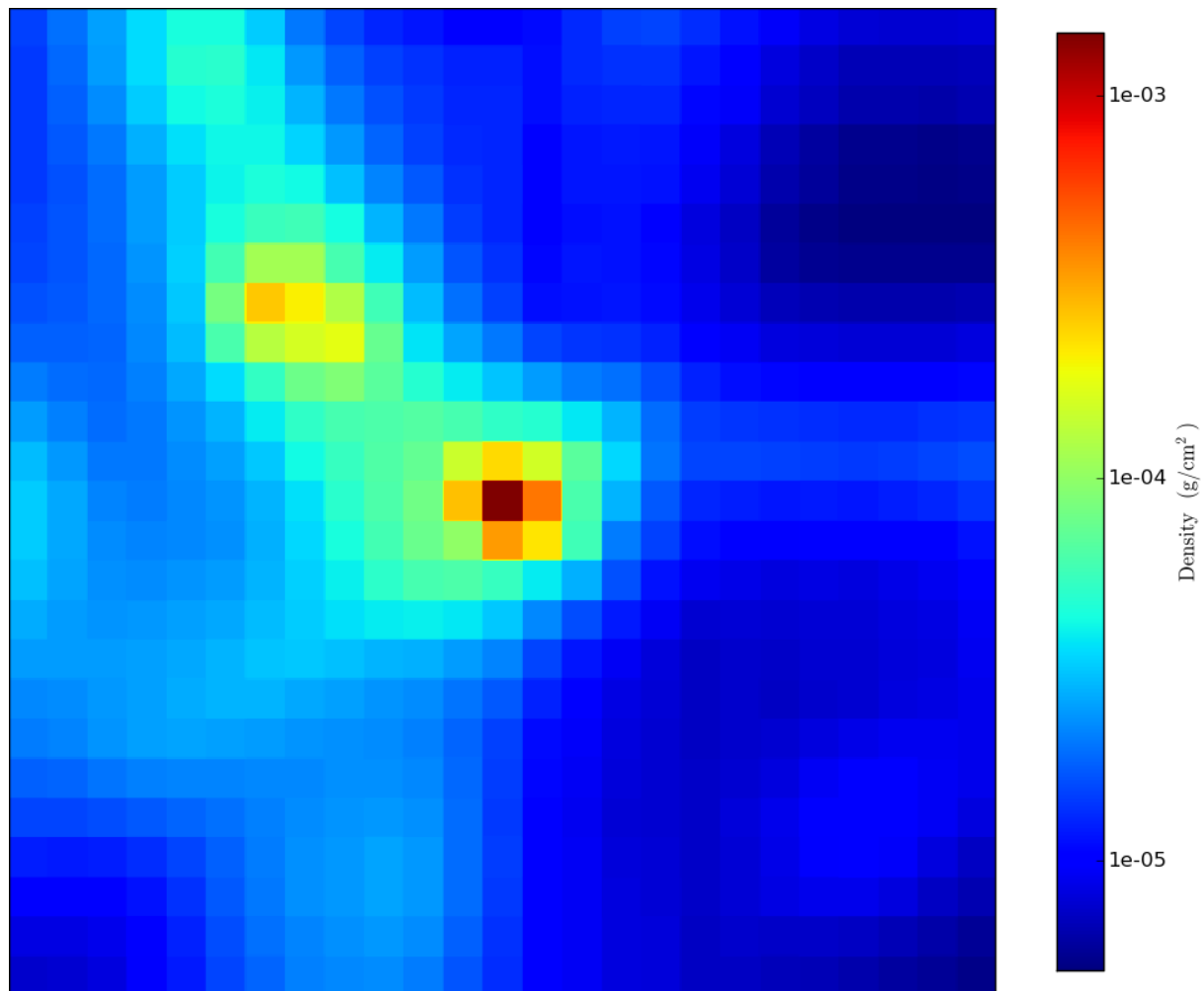
fn = "RedshiftOutput0005" # parameter file to load

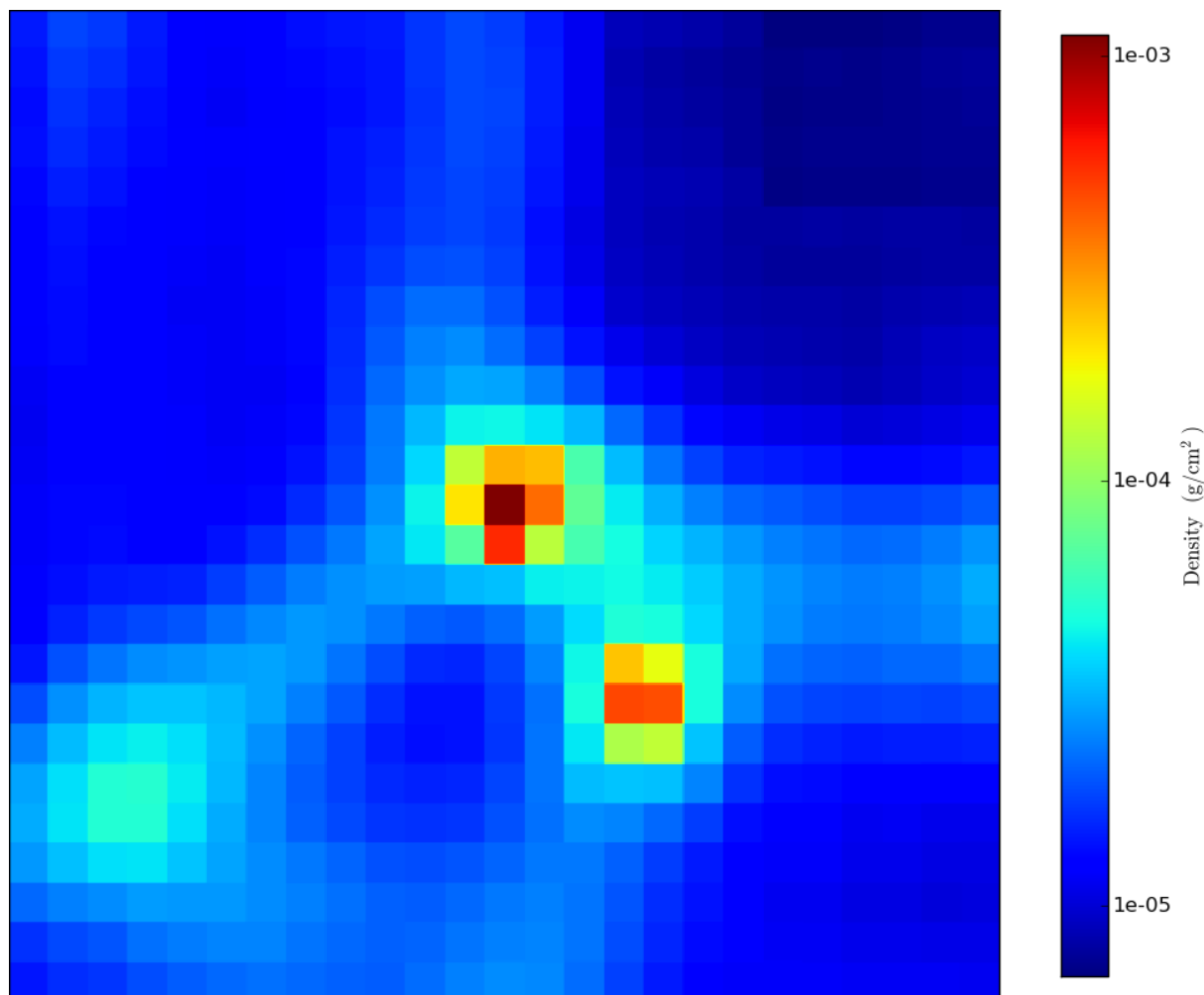
pf = load(fn) # load data
pc = PlotCollection(pf) # defaults to center at most dense point
pc.add_projection("Density", 0, weight="Density") # 0 = x-axis
pc.add_projection("Density", 1, weight="Density") # 1 = y-axis
pc.add_projection("Density", 2, weight="Density") # 2 = z-axis
pc.set_width(1.5, 'mpc') # change width of all plots in pc
pc.save(fn) # save all plots
```



## Sample Output







### 5.3 Aligned cutting plane

This is a recipe to show how to open a dataset, calculate the angular momentum vector in a sphere, and then use that to take an oblique slice.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/aligned\\_cutting\\_plane.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/aligned_cutting_plane.py).

```
from yt.mods import * # set up our namespace

fn = "RedshiftOutput0005" # parameter file to load

pf = load(fn) # load data

# Now let's create a data object that describes the region of which we wish to
# take the angular momentum.

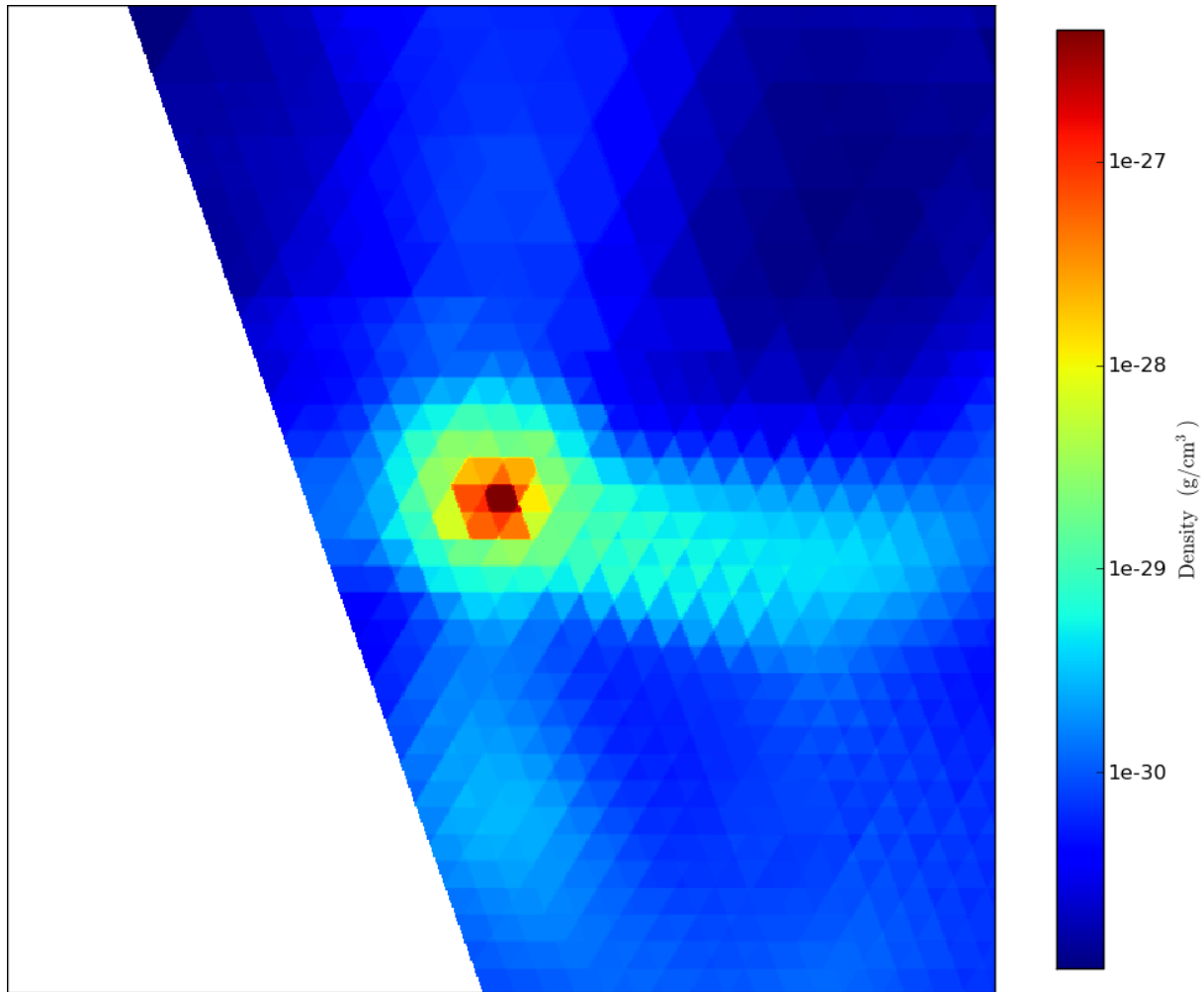
# First find the most dense point, which will serve as our center. We get the
# most dense value for free, too! This is an operation on the 'hierarchy',
# rather than the parameter file.
v, c = pf.h.find_max("Density")
print "Found highest density of %0.3e at %s" % (v, c)
```

```
# Now let's get a sphere centered at this most dense point, c. We have to
# convert '5' into Mpc, which means we have to use unit conversions provided by
# the parameter file. To convert *into* code units, we divide. (To convert
# back, we multiply.)
sp = pf.h.sphere(c, 5.0 / pf["mpc"])
# Now we have an object which contains all of the data within 5 megaparsecs of
# the most dense point. So we want to calculate the angular momentum vector of
# this 5 Mpc set of gas, and yt provides the facility for that inside a
# "derived quantity" property. So we use that, and it returns a vector.
L = sp.quantities["AngularMomentumVector"]()

print "Angular momentum vector: %s" % (L)

pc = PlotCollection(pf, center=c) # Make a new plot holder
pc.add_cutting_plane("Density", L) # Add our oblique slice
pc.set_width(2.5, 'mpc') # change the width
pc.save(fn) # save out with the pf as a prefix to the image name
```

## Sample Output



## 5.4 Sum mass in sphere

This recipe shows how to take a sphere, centered on the most dense point, and sum up the total mass in baryons and particles within that sphere. Note that this recipe will take advantage of multiple CPUs if executed with mpirun and supplied the `-parallel` command line argument.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/sum\\_mass\\_in\\_sphere.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/sum_mass_in_sphere.py).

```
from yt.mods import * # set up our namespace

fn = "RedshiftOutput0005" # parameter file to load

pf = load(fn) # load data
v, c = pf.h.find_max("Density")
sp = pf.h.sphere(c, 1.0/pf["mpc"])

baryon_mass, particle_mass = sp.quantities["TotalQuantity"](
    ["CellMassMsun", "ParticleMassMsun"], lazy_reader=True)
```

```
print "Total mass in sphere is %0.5e (gas = %0.5e / particles = %0.5e)" % \
      (baryon_mass + particle_mass, baryon_mass, particle_mass)
```

## 5.5 Simple phase

This is a simple recipe to show how to open a dataset and then plot a phase plot showing mass distribution in the rho-T plane.

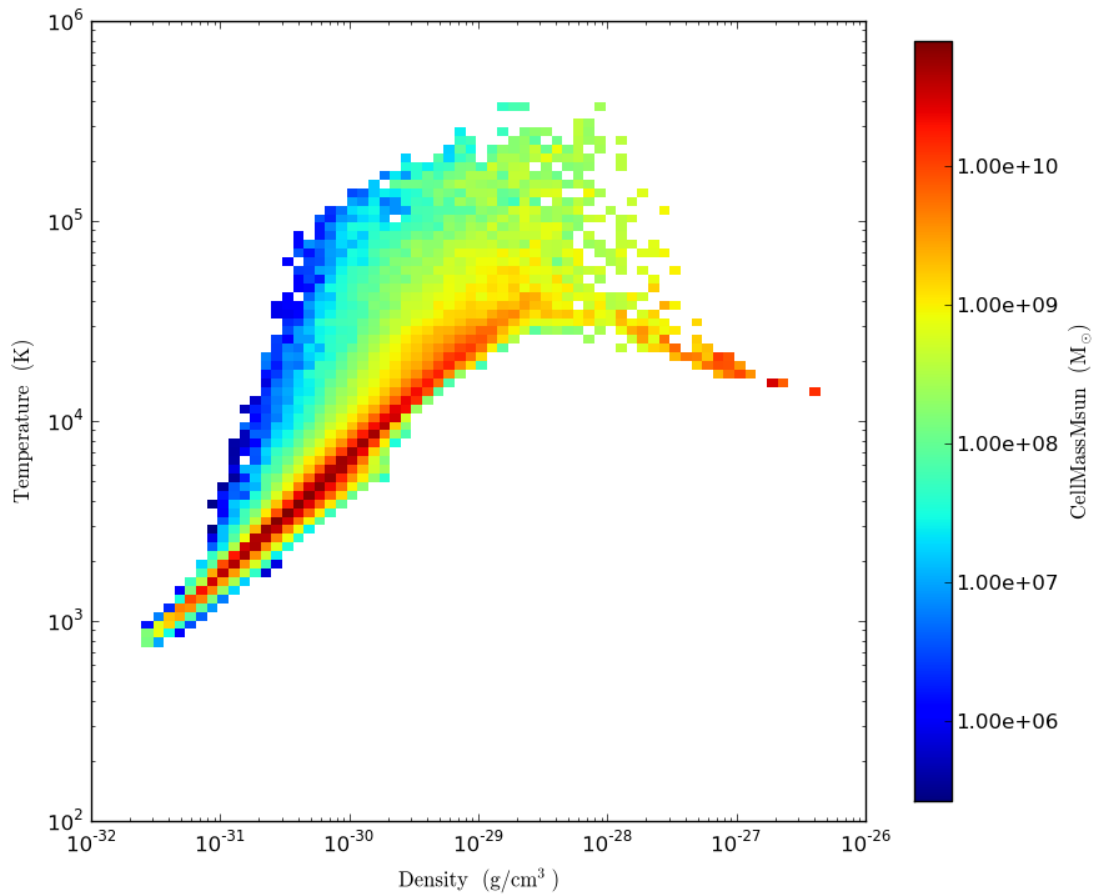
The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/simple\\_phase.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/simple_phase.py).

```
from yt.mods import * # set up our namespace

fn = "RedshiftOutput0005" # parameter file to load

pf = load(fn) # load data
pc = PlotCollection(pf) # defaults to center at most dense point
pc.add_phase_sphere(10.0, "mpc", # how many of which unit at pc.center
    ["Density", "Temperature", "CellMassMsun"], # our fields: x, y, color
    weight=None) # don't take the average value in a cell, just sum them up
pc.save(fn) # save all plots
```

## Sample Output



## 5.6 Simple profile

This is a simple recipe to show how to open a dataset and then plot a profile showing mass-weighted average Temperature as a function of Density inside a sphere.

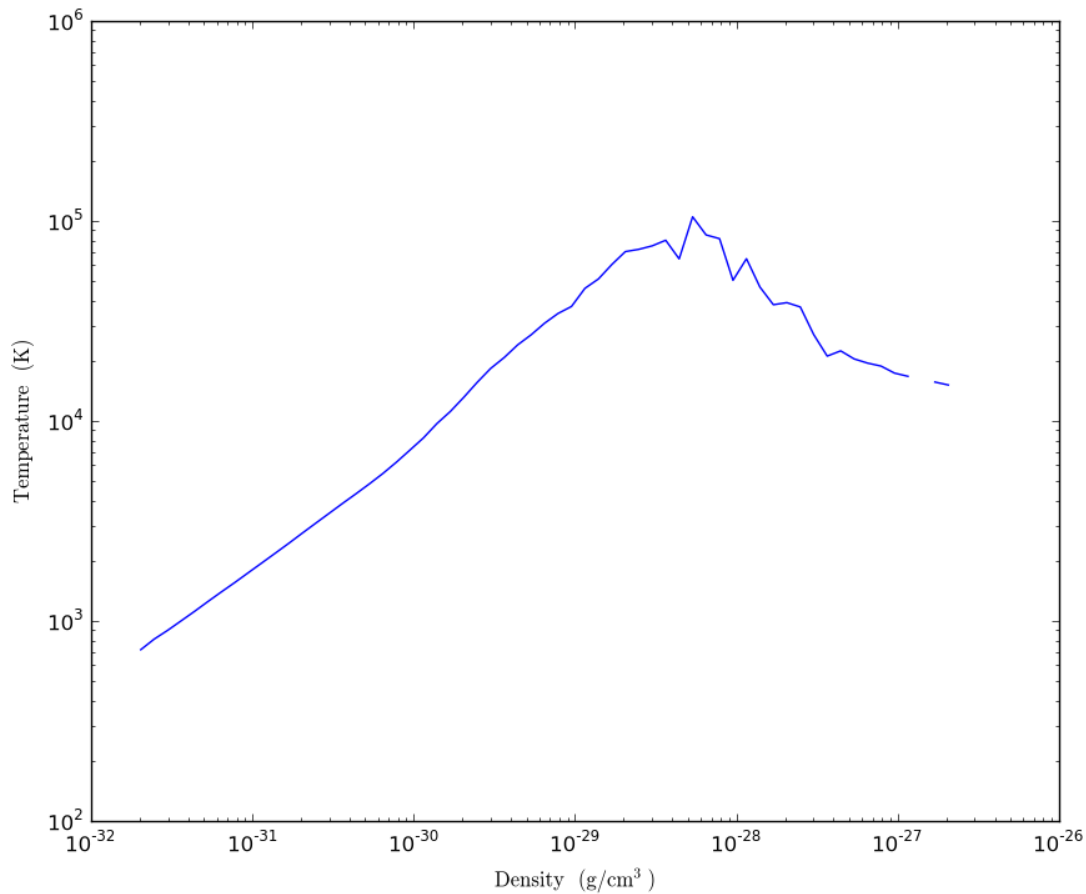
The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/simple\\_profile.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/simple_profile.py).

```
from yt.mods import * # set up our namespace

fn = "RedshiftOutput0005" # parameter file to load

pf = load(fn) # load data
pc = PlotCollection(pf) # defaults to center at most dense point
pc.add_profile_sphere(10.0, "mpc", # how many of which unit at pc.center
    ["Density", "Temperature"], weight="CellMassMsun") # x, y, weight
pc.save(fn) # save all plots
```

## Sample Output



## 5.7 Simple radial profile

This is a simple recipe to show how to open a dataset and then plot a radial profile showing mass-weighted average Density inside a sphere.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/simple\\_radial\\_profile.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/simple_radial_profile.py).

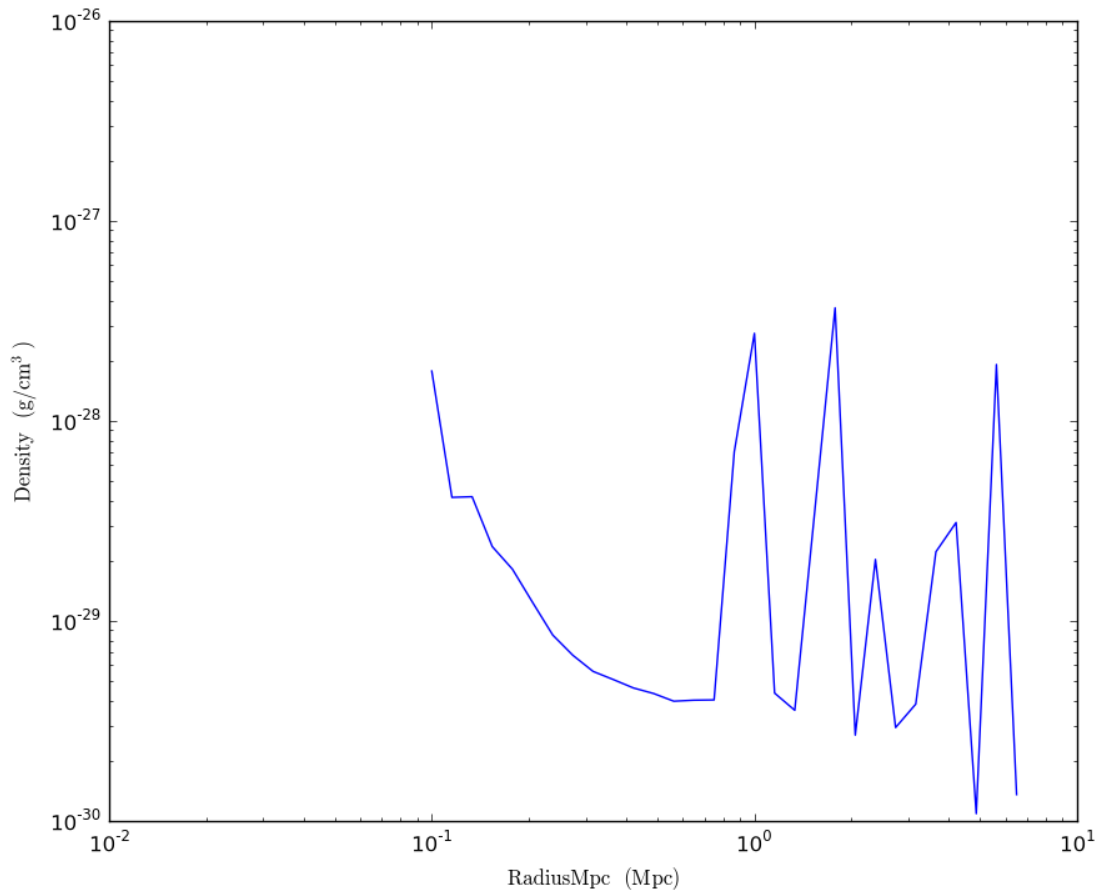
```
from yt.mods import * # set up our namespace

fn = "RedshiftOutput0005" # parameter file to load

pf = load(fn) # load data
pc = PlotCollection(pf) # defaults to center at most dense point
pc.add_profile_sphere(10.0, "mpc", # how many of which unit at pc.center
    ["RadiusMpc", "Density"], weight="CellMassMsun", # x, y, weight
    x_bounds = (1e-3, 10.0)) # cut out zero-radius and tiny-radius cells
    # But ... weight defaults to CellMassMsun, so we're being redundant here!
pc.save(fn) # save all plots
```



## Sample Output



## 5.8 Halo finding

This script shows the simple way of getting halo information.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/halo\\_finding.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/halo_finding.py).

```
from yt.mods import * # set up our namespace

fn = "RedshiftOutput0005" # parameter file to load

pf = load(fn) # load data
halos = HaloFinder(pf)
halos.write_out("%s_halos.txt" % pf)
```

## Sample Output

RedshiftOutput0005\_halos.txt

```
# HALOS FOUND WITH HOP
# Group      Mass      # part      max densx      y      z      center-of-mass      x
0      3.349212591e+11      822      8.530865463e+03      9.474301815e-01
1      2.758414750e+11      677      8.202446551e+03      1.796526850e-01
2      1.947595643e+11      478      4.555572625e+03      5.532828495e-01
3      1.377170141e+11      338      3.242180406e+03      9.754010795e-01
4      1.075659518e+11      264      2.767986753e+03      9.062823292e-01
5      8.311914461e+10      204      2.716905006e+03      7.370274945e-01
6      7.985957031e+10      196      3.728141647e+03      7.032178905e-01
7      7.374786850e+10      181      2.707414503e+03      8.504477878e-01
8      5.419042271e+10      133      2.289804554e+03      3.776429757e-01
9      5.256063556e+10      129      1.414660466e+03      8.358023677e-02
10     3.381808334e+10      83      7.426291275e+02      7.586783008e-01
11     3.218829620e+10      79      1.065896463e+03      7.922319010e-01
12     2.770638154e+10      68      6.274043905e+02      8.762047753e-01
13     2.648404117e+10      65      8.422493962e+02      7.145945477e-01
14     2.526170081e+10      62      5.496262872e+02      2.661371336e-01
15     2.363191366e+10      58      6.261403270e+02      5.537976778e-01
16     2.240957330e+10      55      7.216382764e+02      2.869443986e-01
17     2.077978615e+10      51      7.207463353e+02      7.925281425e-01
18     1.996489258e+10      49      5.250658436e+02      8.289750280e-01
19     1.955744579e+10      48      4.827141235e+02      8.332161433e-01
```

## 5.9 Arbitrary vectors on slice

This is a simple recipe to show how to open a dataset, plot a slice through it, and add some extra vectors on top. Here we've used the imaginary fields `magnetic_field_x`, `magnetic_field_y` and `magnetic_field_z`.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/arbitrary\\_vectors\\_on\\_slice.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/arbitrary_vectors_on_slice.py).

```
from yt.mods import * # set up our namespace

fn = "RedshiftOutput0005" # parameter file to load
ax = 0 # x-axis

pf = load(fn) # load data
pc = PlotCollection(pf) # defaults to center at most dense point
p = pc.add_slice("Density", ax)
v1 = "magnetic_field_%s" % (axis_names[x_dict[ax]])
v2 = "magnetic_field_%s" % (axis_names[y_dict[ax]])
p.modify["quiver"](v1, v2) # This takes a few arguments, but we'll use the defaults
                           # here. You can control the 'skip' factor in the
                           # vectors.
pc.set_width(2.5, 'mpc') # change width of all plots in pc
pc.save(fn) # save all plots
```

## 5.10 Contours on slice

This is a simple recipe to show how to open a dataset, plot a slice through it, and add contours of another quantity on top.

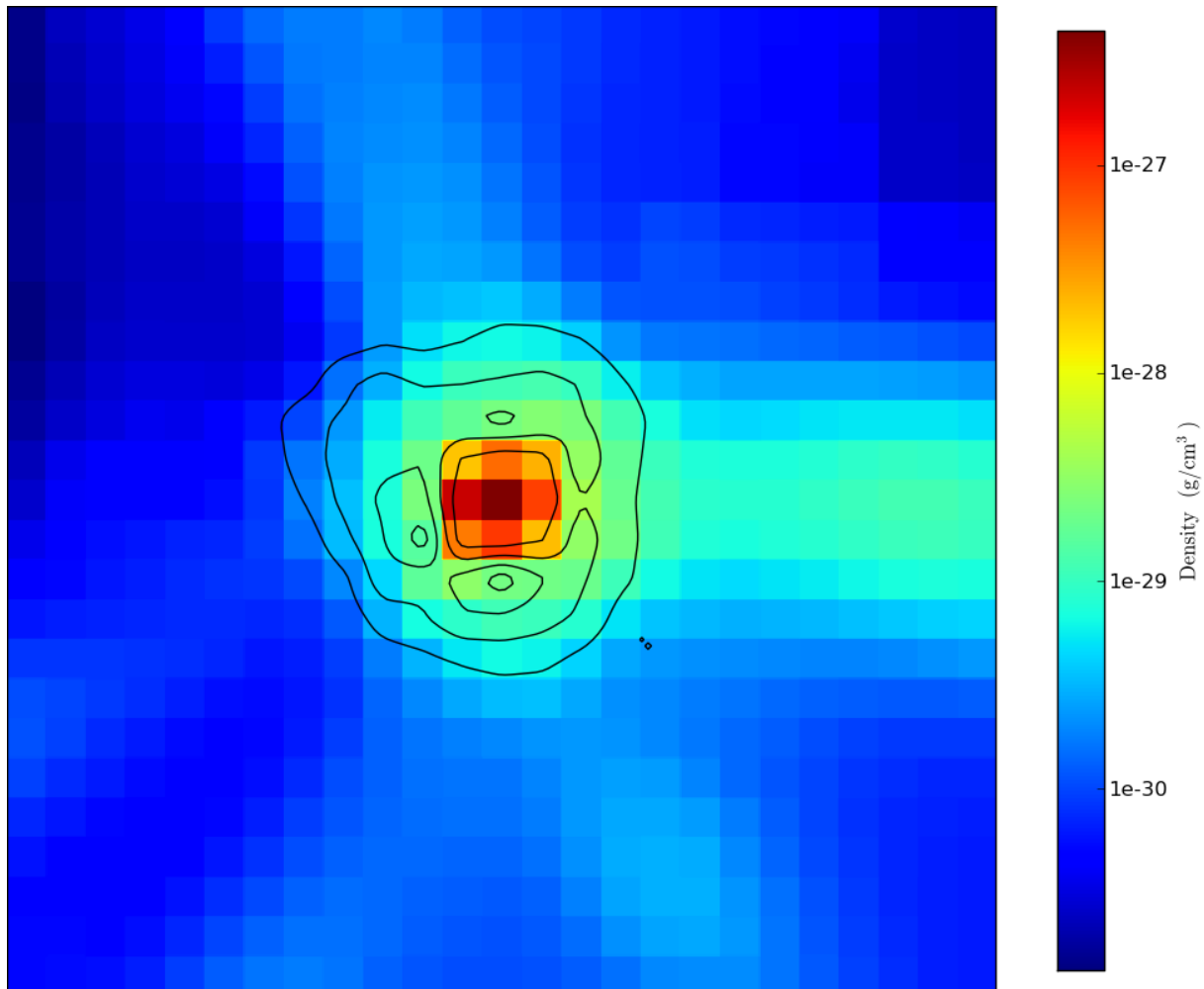
The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/contours\\_on\\_slice.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/contours_on_slice.py).

```
from yt.mods import * # set up our namespace

fn = "RedshiftOutput0005" # parameter file to load

pf = load(fn) # load data
pc = PlotCollection(pf) # defaults to center at most dense point
p = pc.add_slice("Density", 0) # 0 = x-axis
p.modify["contour"]("Temperature")
pc.set_width(1.5, 'mpc') # change width of all plots in pc
pc.save(fn) # save all plots
```

## Sample Output



## 5.11 Velocity vectors on slice

This is a simple recipe to show how to open a dataset, plot a slice through it, and add velocity vectors on top.

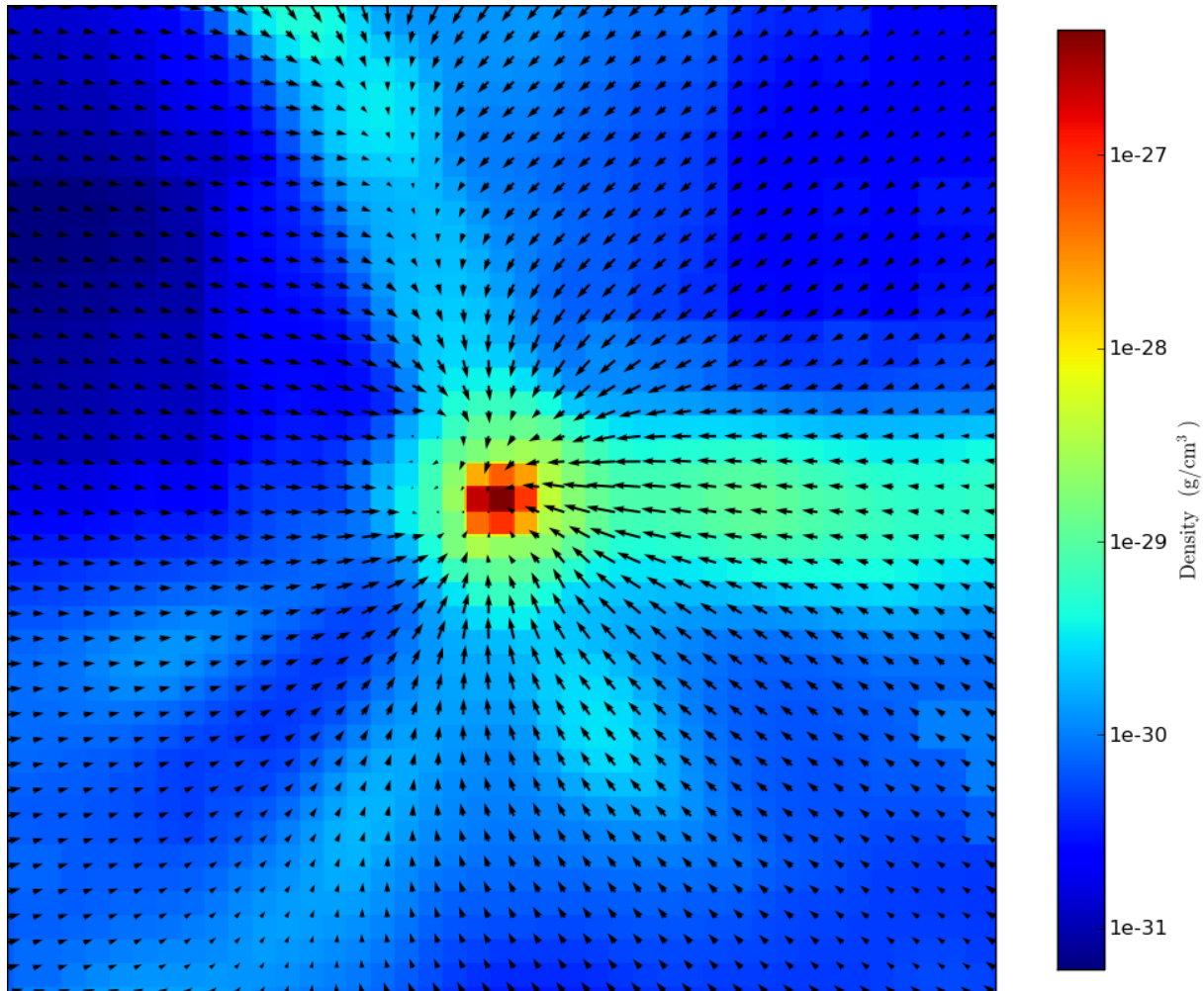
The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/velocity\\_vectors\\_on\\_slice.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/velocity_vectors_on_slice.py).

```
from yt.mods import * # set up our namespace

fn = "RedshiftOutput0005" # parameter file to load

pf = load(fn) # load data
pc = PlotCollection(pf) # defaults to center at most dense point
p = pc.add_slice("Density", 0) # 0 = x-axis
p.modify["velocity"]() # This takes a few arguments, but we'll use the defaults
                        # here. You can control the 'skip' factor in the
                        # vectors.
pc.set_width(2.5, 'mpc') # change width of all plots in pc
pc.save(fn) # save all plots
```

## Sample Output



## 5.12 Average value

This recipe finds the average value of a quantity through the entire box. Note that this recipe will take advantage of multiple CPUs if executed with `mpirun` and supplied the `-parallel` command line argument.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/average\\_value.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/average_value.py).

```
from yt.mods import *

fn = "RedshiftOutput0005" # parameter file to load
pf = load(fn) # load data

field = "Temperature" # The field to average
weight = "CellMassMsun" # The weight for the average

dd = pf.h.all_data() # This is a region describing the entire box,
                     # but note it doesn't read anything in yet!
# We now use our 'quantities' call to get the average quantity
average_value = dd.quantities["WeightedAverageQuantity"]()
```

```
field, weight, lazy_reader=True)

print "Average %s (weighted by %s) is %0.5e" % (field, weight, average_value)
```

## 5.13 Find clumps

This is a recipe to show how to find topologically connected sets of cells inside a dataset. It returns these clumps and they can be inspected or visualized as would any other data object. More detail on this method can be found in [astro-ph/0806.1653](http://astro-ph/0806.1653).

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/find\\_clumps.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/find_clumps.py).

```
from yt.mods import * # set up our namespace

fn = "RedshiftOutput0005" # parameter file to load
field = "Density" # this is the field we look for contours over -- we could do
                  # this over anything. Other common choices are 'AveragedDensity'
                  # and 'Dark_Matter_Density'.
step = 10.0 # This is the multiplicative interval between contours.

pf = load(fn) # load data

# We want to find clumps over the entire dataset, so we'll just grab the whole
# thing! This is a convenience parameter that prepares an object that covers
# the whole domain. Note, though, that it will load on demand and not before!
data_source = pf.h.all_data()

# Now we set some sane min/max values between which we want to find contours.
# This is how we tell the clump finder what to look for -- it won't look for
# contours connected below or above these threshold values.
c_min = 10**na.floor(na.log10(data_source[field]).min() )
c_max = 10**na.floor(na.log10(data_source[field]).max()+1)

# Now find get our 'base' clump -- this one just covers the whole domain.
master_clump = Clump(data_source, None, field)

# This next command accepts our base clump and we say the range between which
# we want to contour. It recursively finds clumps within the master clump, at
# intervals defined by the step size we feed it. The current value is
# *multiplied* by step size, rather than added to it -- so this means if you
# want to look in log10 space intervals, you would supply step = 10.0.
find_clumps(master_clump, c_min, c_max, step)

# As it goes, it appends the information about all the sub-clumps to the
# master-clump. Among different ways we can examine it, there's a convenience
# function for outputting the full hierarchy to a file.
f = open('%s_clump_hierarchy.txt' % pf, 'w')
write_clump_hierarchy(master_clump, 0, f)
f.close()

# We can also output some handy information, as well.
f = open('%s_clumps.txt' % pf, 'w')
write_clumps(master_clump, 0, f)
f.close()
```

---

```
# If you'd like to visualize these clumps, a list of clumps can be supplied to
# the "clumps" callback on a plot.
```

## Sample Output

```
RedshiftOutput0005_clump_hierarchy.txt
```

```
Clump at level 0:
Cells: 714638
Mass: 2.867994e+12 Msolar
Jeans Mass (vol-weighted): 1.945683e+10 Msolar
Jeans Mass (mass-weighted): 4.959227e+10 Msolar
Max grid level: 2
Min number density: 7.299408e-09 cm^-3
Max number density: 1.594844e-03 cm^-3

    Clump at level 1:
    Cells: 136
    Mass: 3.150459e+10 Msolar
    Jeans Mass (vol-weighted): 2.313084e+11 Msolar
    Jeans Mass (mass-weighted): 6.949434e+10 Msolar
    Max grid level: 2
    Min number density: 3.599559e-06 cm^-3
    Max number density: 4.158763e-04 cm^-3

        Clump at level 2:
        Cells: 1
        Mass: 3.698225e+09 Msolar
        Jeans Mass (vol-weighted): 3.532331e+09 Msolar
        Jeans Mass (mass-weighted): 3.532331e+09 Msolar
        Max grid level: 2
        Min number density: 4.144234e-04 cm^-3
        Max number density: 4.144234e-04 cm^-3

    Clump at level 1:
    Cells: 944
    Mass: 1.363021e+11 Msolar
    Jeans Mass (vol-weighted): 3.786709e+11 Msolar
    Jeans Mass (mass-weighted): 1.977858e+11 Msolar
    Max grid level: 2
    Min number density: 3.593325e-06 cm^-3
    Max number density: 7.125931e-04 cm^-3

        Clump at level 2:
        Cells: 2
        Mass: 1.205801e+10 Msolar
        Jeans Mass (vol-weighted): 2.555663e+09 Msolar
        Jeans Mass (mass-weighted): 2.555660e+09 Msolar
        Max grid level: 2
        Min number density: 6.737547e-04 cm^-3
        Max number density: 6.774670e-04 cm^-3
```

```
RedshiftOutput0005_clumps.txt
```

```
Clump:
Cells: 1
Mass: 3.698225e+09 Msolar
Jeans Mass (vol-weighted): 3.532331e+09 Msolar
Jeans Mass (mass-weighted): 3.532331e+09 Msolar
Max grid level: 2
Min number density: 4.144234e-04 cm-3
Max number density: 4.144234e-04 cm-3
```

```
Clump:
Cells: 2
Mass: 1.205801e+10 Msolar
Jeans Mass (vol-weighted): 2.555663e+09 Msolar
Jeans Mass (mass-weighted): 2.555660e+09 Msolar
Max grid level: 2
Min number density: 6.737547e-04 cm-3
Max number density: 6.774670e-04 cm-3
```

## 5.14 Global phase plots

This is a simple recipe to show how to open a dataset and then plot a couple phase diagrams, save them, and quit. Note that this recipe will take advantage of multiple CPUs if executed with mpirun and supplied the `-parallel` command line argument.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/global\\_phase\\_plots.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/global_phase_plots.py).

```
from yt.mods import * # set up our namespace

fn = "RedshiftOutput0005" # parameter file to load

pf = load(fn) # load data
dd = pf.h.all_data() # This is an object that describes the entire box
pc = PlotCollection(pf) # defaults to center at most dense point

# We plot the average x-velocity (mass-weighted) in our object as a function of
# Electron_Density and Temperature
plot=pc.add_phase_object(dd, ["Electron_Density", "Temperature", "x-velocity"])
    lazy_reader = True)

# We now plot the average value of x-velocity as a function of temperature
plot=pc.add_profile_object(dd, ["Temperature", "x-velocity"],
    lazy_reader = True)

# Finally, the average electron density as a function of the magnitude of the
# velocity
plot=pc.add_profile_object(dd, ["Electron_Density", "VelocityMagnitude"],
    lazy_reader = True)
pc.save() # save all plots
```



## 5.15 Halo mass info

This recipe finds halos and then prints out information about them. Note that this recipe will take advantage of multiple CPUs if executed with `mpirun` and supplied the `-parallel` command line argument.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/halo\\_mass\\_info.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/halo_mass_info.py).

```
from yt.mods import *

fn = "RedshiftOutput0005" # parameter file to load
pf = load(fn) # load data

# First we run our halo finder to identify all the halos in the dataset. This
# can take arguments, but the default are pretty sane.
halos = HaloFinder(pf)

f = open("%s_halo_info.txt" % pf, "w")

# Now, for every halo, we get the baryon data and examine it.
for halo in halos:
    # The halo has a property called 'get_sphere' that obtains a sphere
    # centered on the point of maximum density (or the center of mass, if that
    # argument is supplied) and with the radius the maximum particle radius of
    # that halo.
    sphere = halo.get_sphere()
    # We use the quantities[] method to get the total mass in baryons and in
    # particles.
    baryon_mass, particle_mass = sphere.quantities["TotalQuantity"](
        ["CellMassMsun", "ParticleMassMsun"], lazy_reader=True)
    # Now we print out this information, along with the ID.
    f.write("Total mass in HOP group %s is %0.5e (gas = %0.5e / particles = %0.5e)\n" % \
        (halo.id, baryon_mass + particle_mass, baryon_mass, particle_mass))
f.close()
```

### Sample Output

RedshiftOutput0005\_halo\_info.txt

```
Total mass in HOP group 0 is 4.83999e+11 (gas = 6.41403e+10 / particles = 4.19859e+11)
Total mass in HOP group 1 is 4.21962e+11 (gas = 6.07831e+10 / particles = 3.61179e+11)
Total mass in HOP group 2 is 2.62190e+11 (gas = 3.86250e+10 / particles = 2.23565e+11)
Total mass in HOP group 3 is 1.95698e+11 (gas = 3.06816e+10 / particles = 1.65016e+11)
Total mass in HOP group 4 is 1.41878e+11 (gas = 2.12738e+10 / particles = 1.20604e+11)
Total mass in HOP group 5 is 1.16476e+11 (gas = 1.82813e+10 / particles = 9.81947e+10)
Total mass in HOP group 6 is 1.24188e+11 (gas = 2.27342e+10 / particles = 1.01454e+11)
Total mass in HOP group 7 is 7.82627e+10 (gas = 1.47010e+10 / particles = 6.35617e+10)
Total mass in HOP group 8 is 6.61199e+10 (gas = 6.63271e+09 / particles = 5.94872e+10)
Total mass in HOP group 9 is 7.63012e+10 (gas = 1.27395e+10 / particles = 6.35617e+10)
Total mass in HOP group 10 is 4.28431e+10 (gas = 4.95060e+09 / particles = 3.78926e+10)
Total mass in HOP group 11 is 4.31030e+10 (gas = 3.58066e+09 / particles = 3.95223e+10)
Total mass in HOP group 12 is 2.92851e+10 (gas = 1.17132e+09 / particles = 2.81138e+10)
Total mass in HOP group 13 is 3.13247e+10 (gas = 1.98850e+09 / particles = 2.93362e+10)
Total mass in HOP group 14 is 2.97262e+10 (gas = 7.97481e+08 / particles = 2.89287e+10)
Total mass in HOP group 15 is 2.42277e+10 (gas = 5.95763e+08 / particles = 2.36319e+10)
```

```
Total mass in HOP group 16 is 2.76922e+10 (gas = 1.20811e+09 / particles = 2.64840e+10)
Total mass in HOP group 17 is 2.18954e+10 (gas = 1.11559e+09 / particles = 2.07798e+10)
Total mass in HOP group 18 is 2.09572e+10 (gas = 5.84902e+08 / particles = 2.03723e+10)
Total mass in HOP group 19 is 2.55289e+10 (gas = 2.71190e+09 / particles = 2.28170e+10)
```

## 5.16 Multi width save

This recipe shows a slightly-fancy way to save a couple plots at a lot of different widths, ensuring that across the plots we have the same min/max for the colorbar.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/multi\\_width\\_save.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/multi_width_save.py).

```
from yt.mods import *

fn = "RedshiftOutput0005" # parameter file to load
pf = load(fn) # load data

pc = PlotCollection(pf, center=[0.5, 0.5, 0.5]) # We get our Plot Collection object

# Note that when we save, we will be using string formatting to change all of
# the bits in here. You can add more, or remove some, if you like.
fn = "%(bn)s_%(width)010i_%(unit)s" # template for image file names

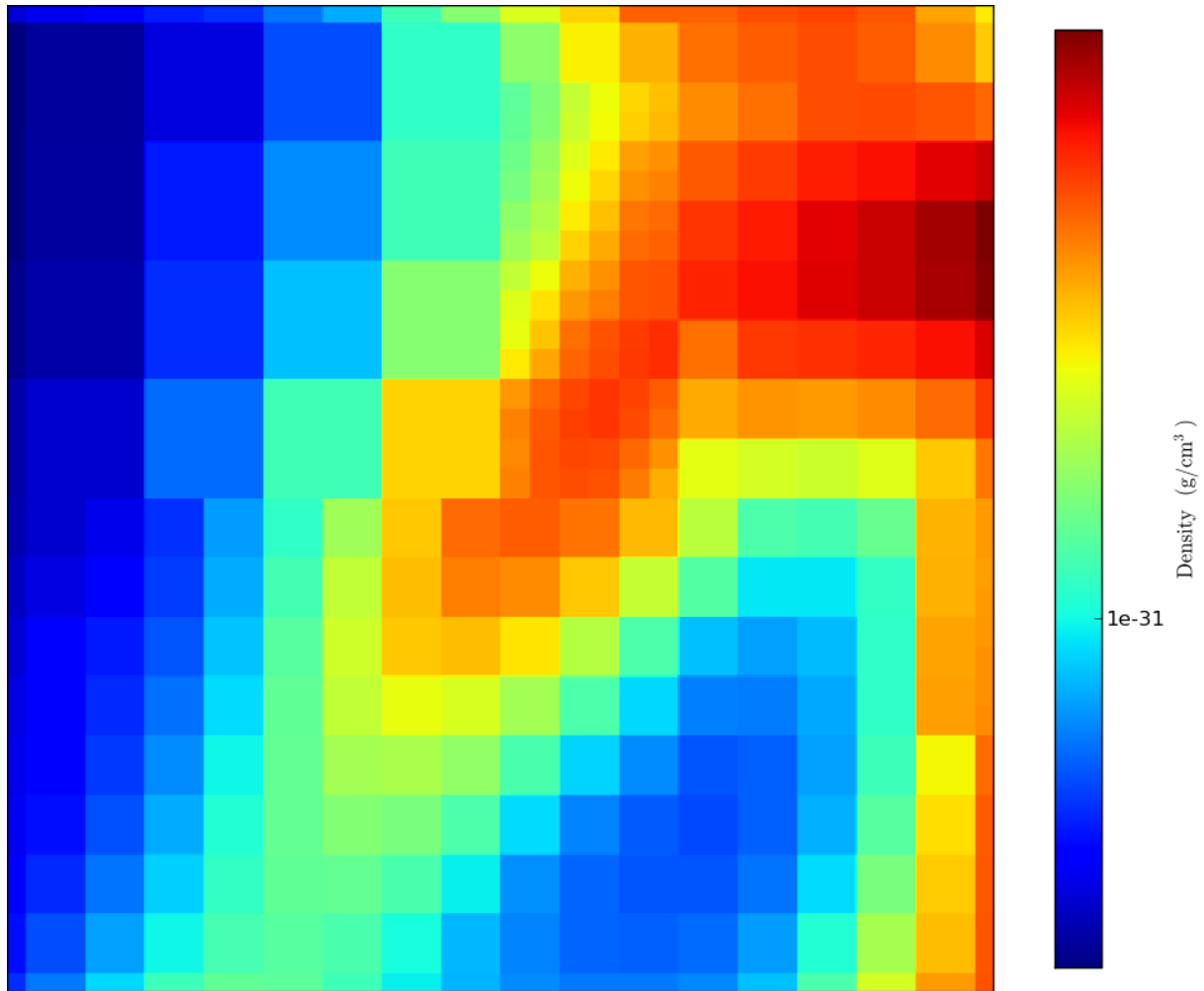
# Now let's set up the widths we want to use.
widths = [ (2, "mpc"), (1000, 'kpc') ]
# We could add on more of these with:
# widths += [ ... ]

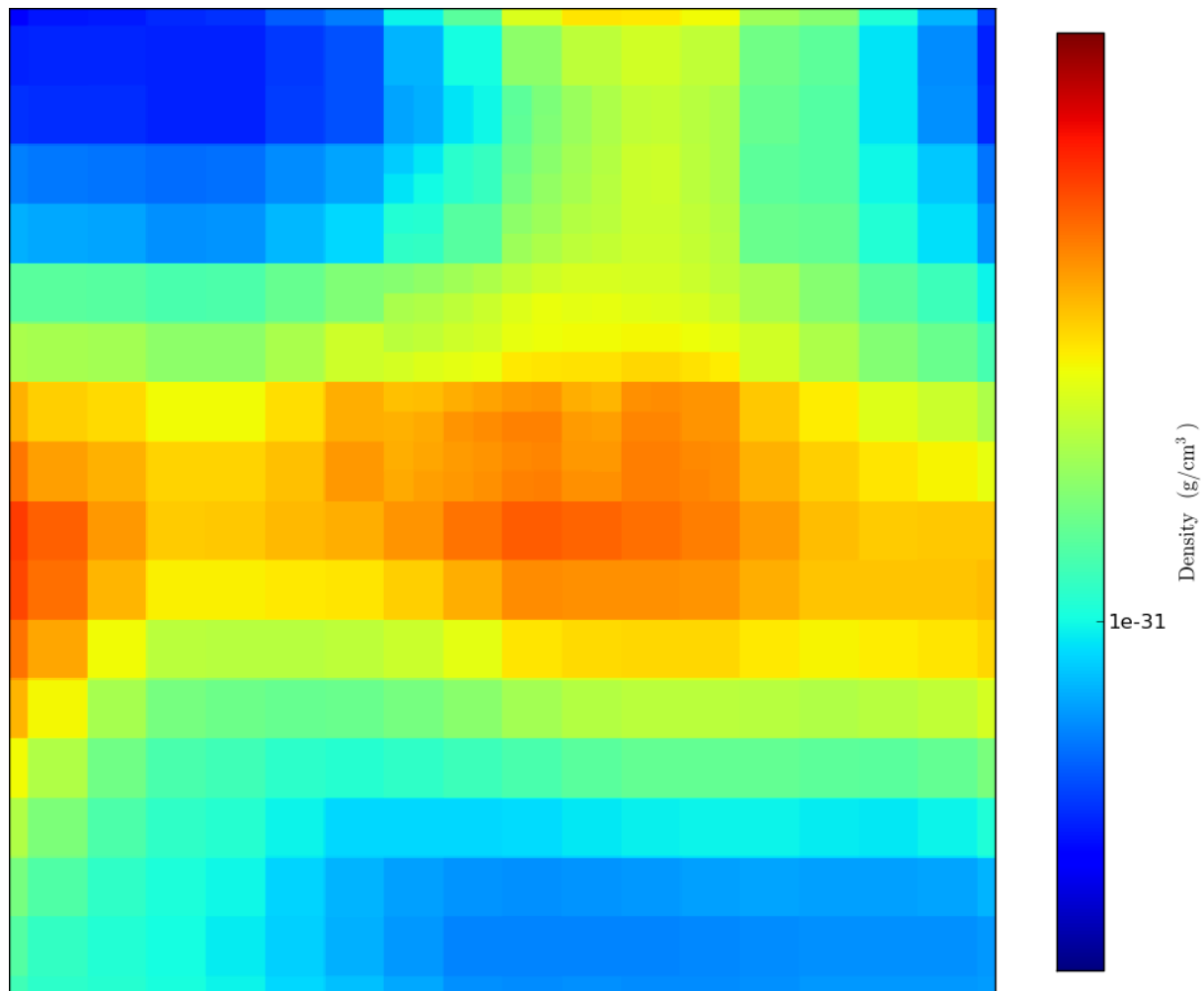
# Now we add a slice for x and y.
pc.add_slice("Density", 0)
pc.add_slice("Density", 1)

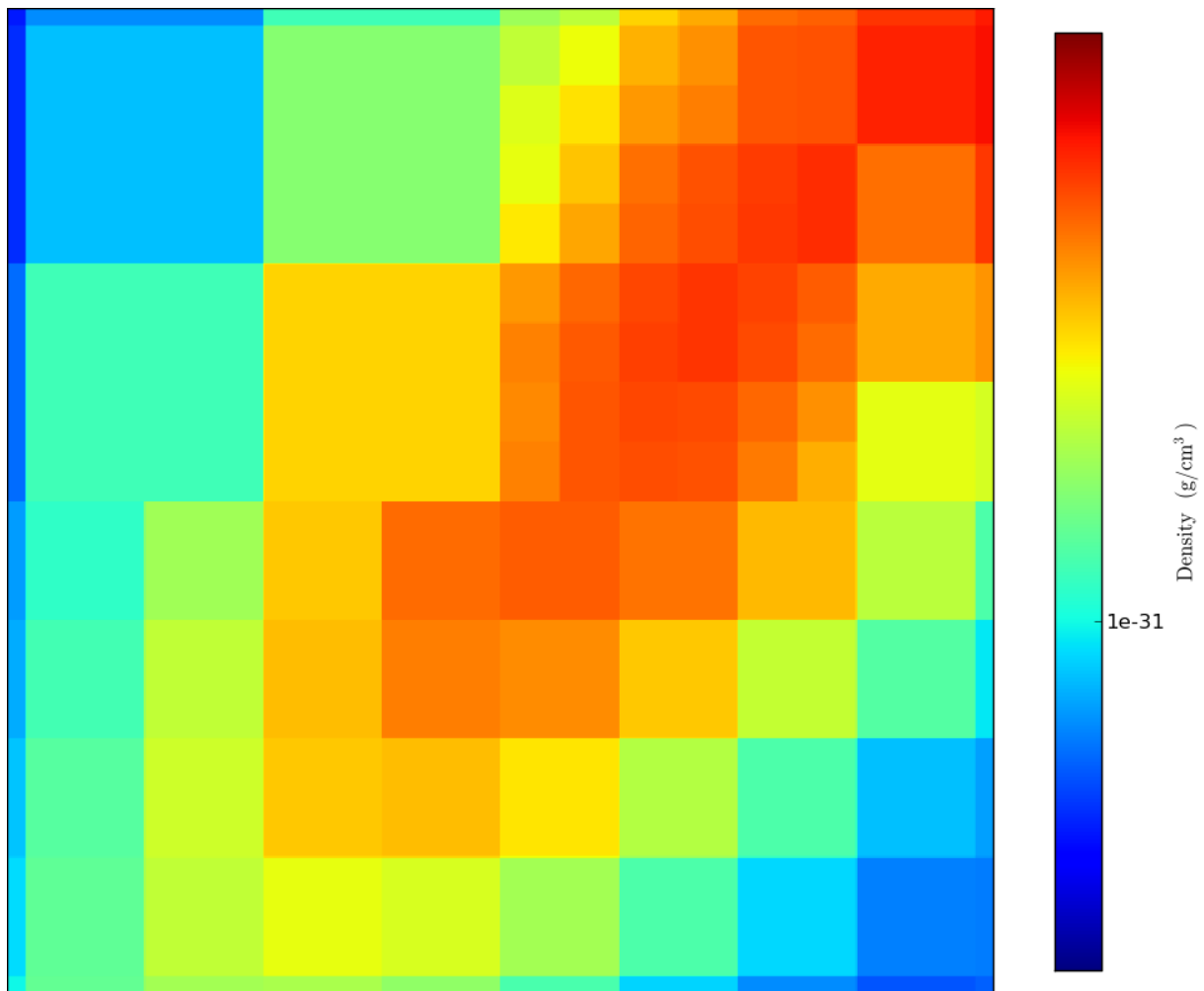
# So for all of our widths, we will set the width of the plot and then make
# sure that our limits for the colorbar are the min/max across the three plots.
# Then we save! Each saved file will have a descriptive name, so we can tell
# them apart.

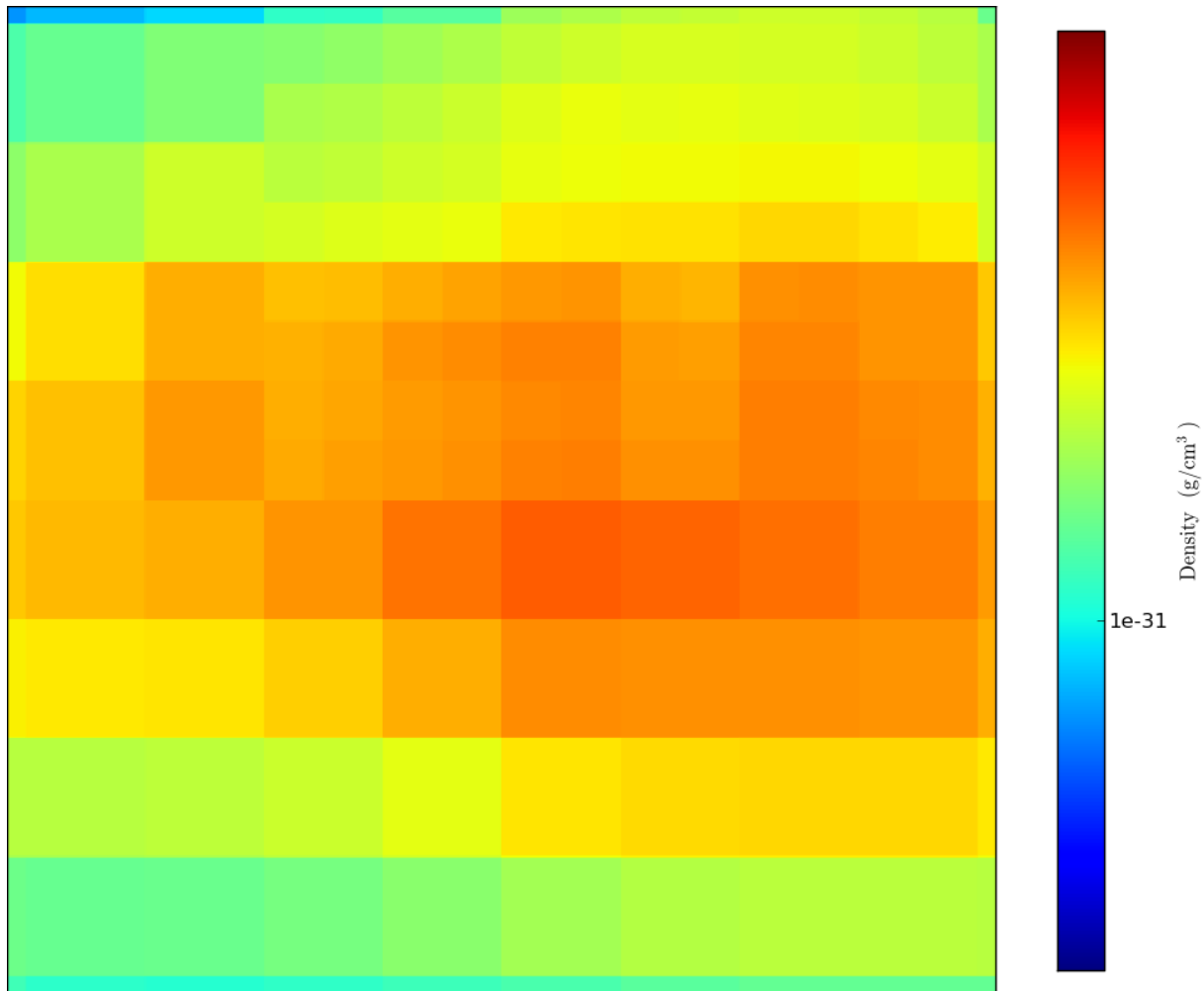
for width, unit in widths:
    pc.set_width(width, unit)
    vmin = min([p.norm.vmin for p in pc.plots])
    vmax = max([p.norm.vmax for p in pc.plots])
    pc.set_zlim(vmin, vmax)
    # This is the string formatting we talked about earlier
    d = {'bn': pf.basename, 'width': width, 'unit': unit}
    pc.save(fn % d)
```

## Sample Output









## 5.17 Zoomin frames

This is a recipe that takes a slice through the most dense point, then creates a bunch of frames as it zooms in. It's important to note that this particular recipe is provided to show how to be more flexible and add annotations and the like – the base system, of a zoomin, is provided by the `yt zoomin` command.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/zoomin\\_frames.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/zoomin_frames.py).

```
from yt.mods import * # set up our namespace

fn = "RedshiftOutput0005" # parameter file to load
n_frames = 5 # This is the number of frames to make -- below, you can see how
               # this is used.
min_dx = 40 # This is the minimum size in smallest_dx of our last frame.
             # Usually it should be set to something like 400, but for THIS
             # dataset, we actually don't have that great of resolution.

pf = load(fn) # load data
frame_template = "frame_%05i" # Template for frame filenames

pc = PlotCollection(pf, center=[0.5, 0.5, 0.5]) # We make a plot collection that defaults to being
```

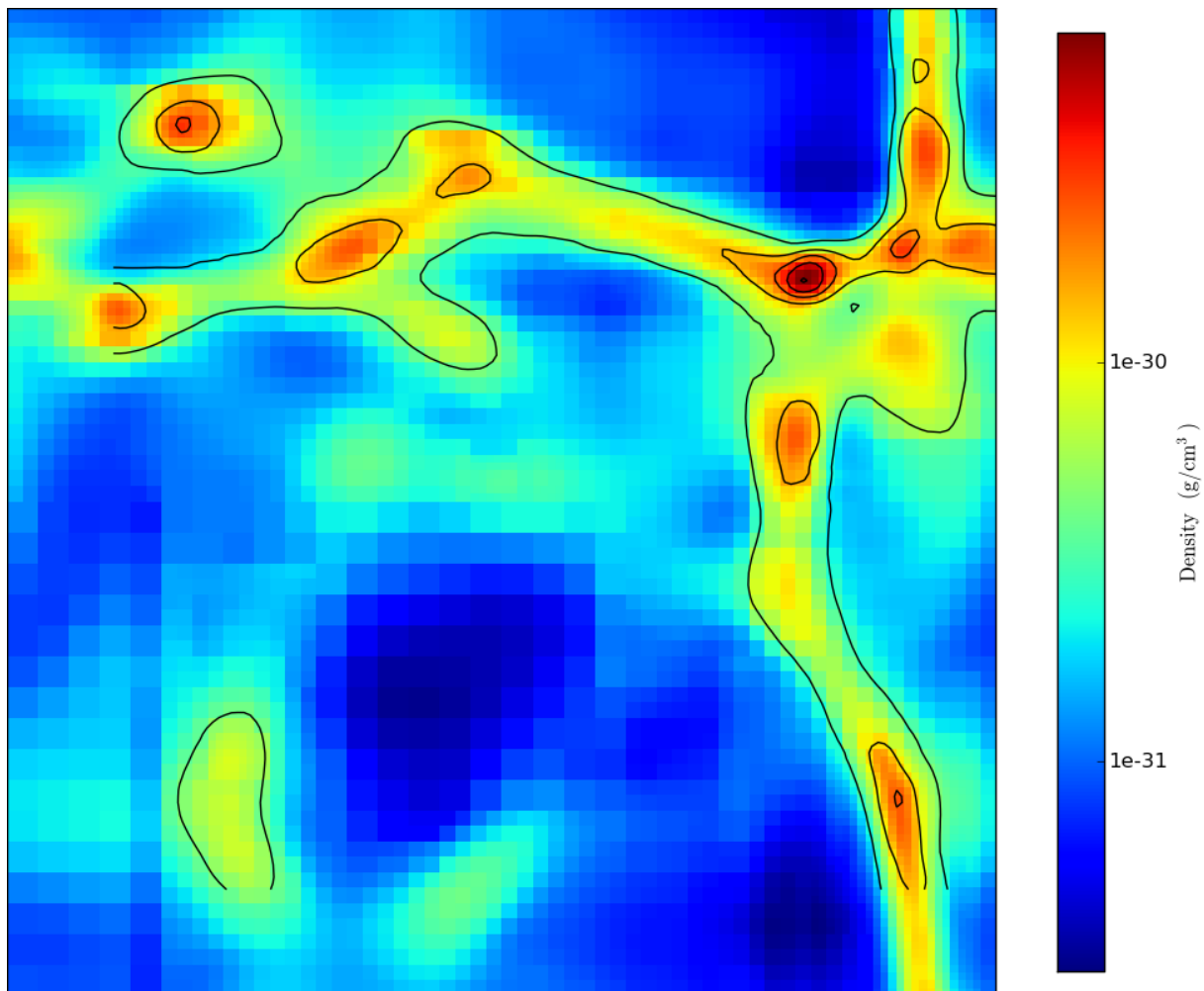
```

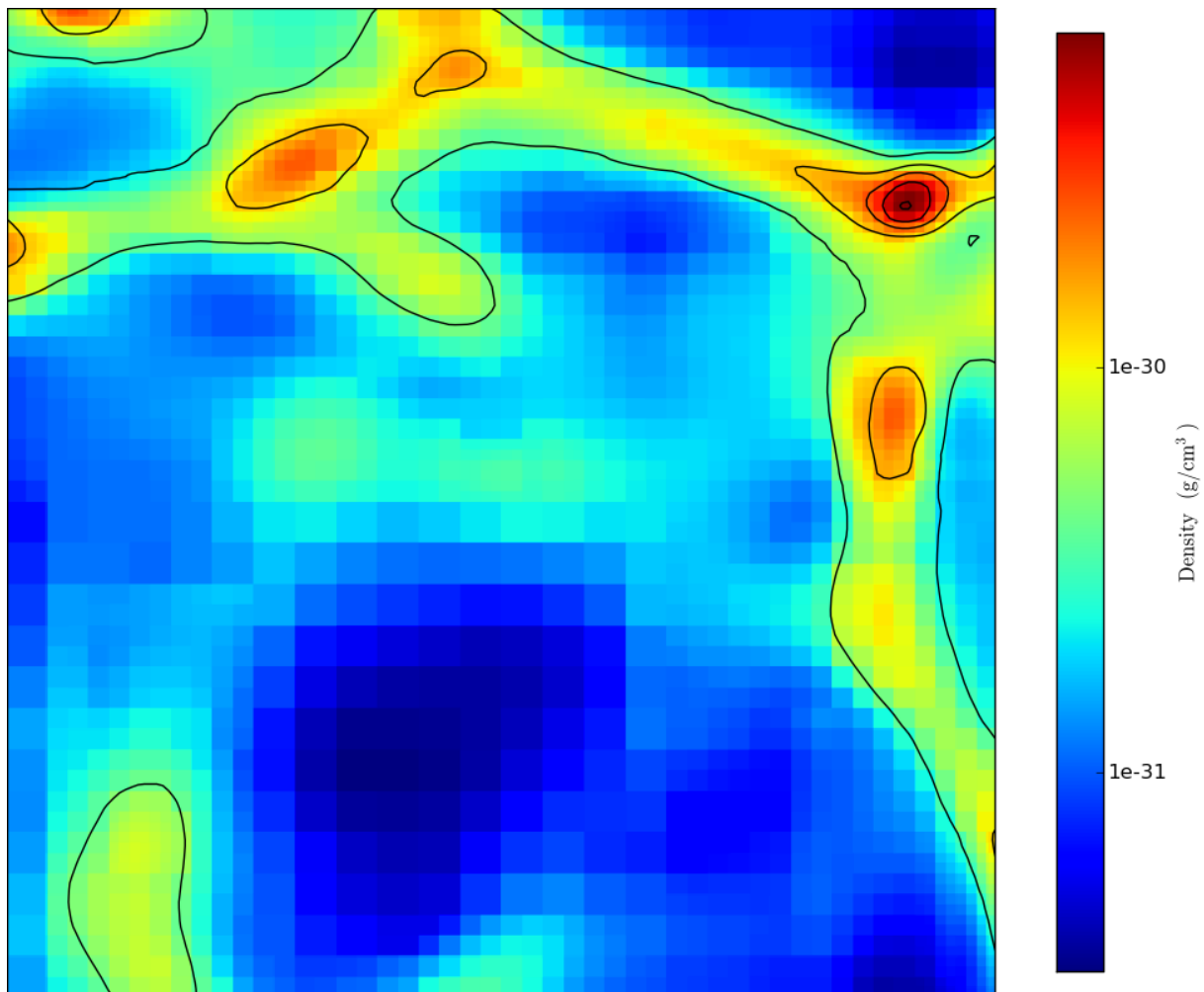
# centered at the most dense point.
p = pc.add_slice("Density", 2) # Add our slice, along z
p.modify["contour"]("Temperature") # We'll contour in temperature -- this kind
# of modification can't be done on the command
# line, so that's why we have the recipe!

# What we do now is a bit fun. "enumerate" returns a tuple for every item --
# the index of the item, and the item itself. This saves us having to write
# something like "i = 0" and then inside the loop "i += 1" for ever loop. The
# argument to enumerate is the 'logspace' function, which takes a minimum and a
# maximum and the number of items to generate. It returns 10^power of each
# item it generates.
for i,v in enumerate(na.logspace(
    0, na.log10(pf.h.get_smallest_dx()*min_dx), n_frames)):
    # We set our width as necessary for this frame ...
    pc.set_width(v,'1')
    # ... and we save!
    pc.save(frame_template % (i))

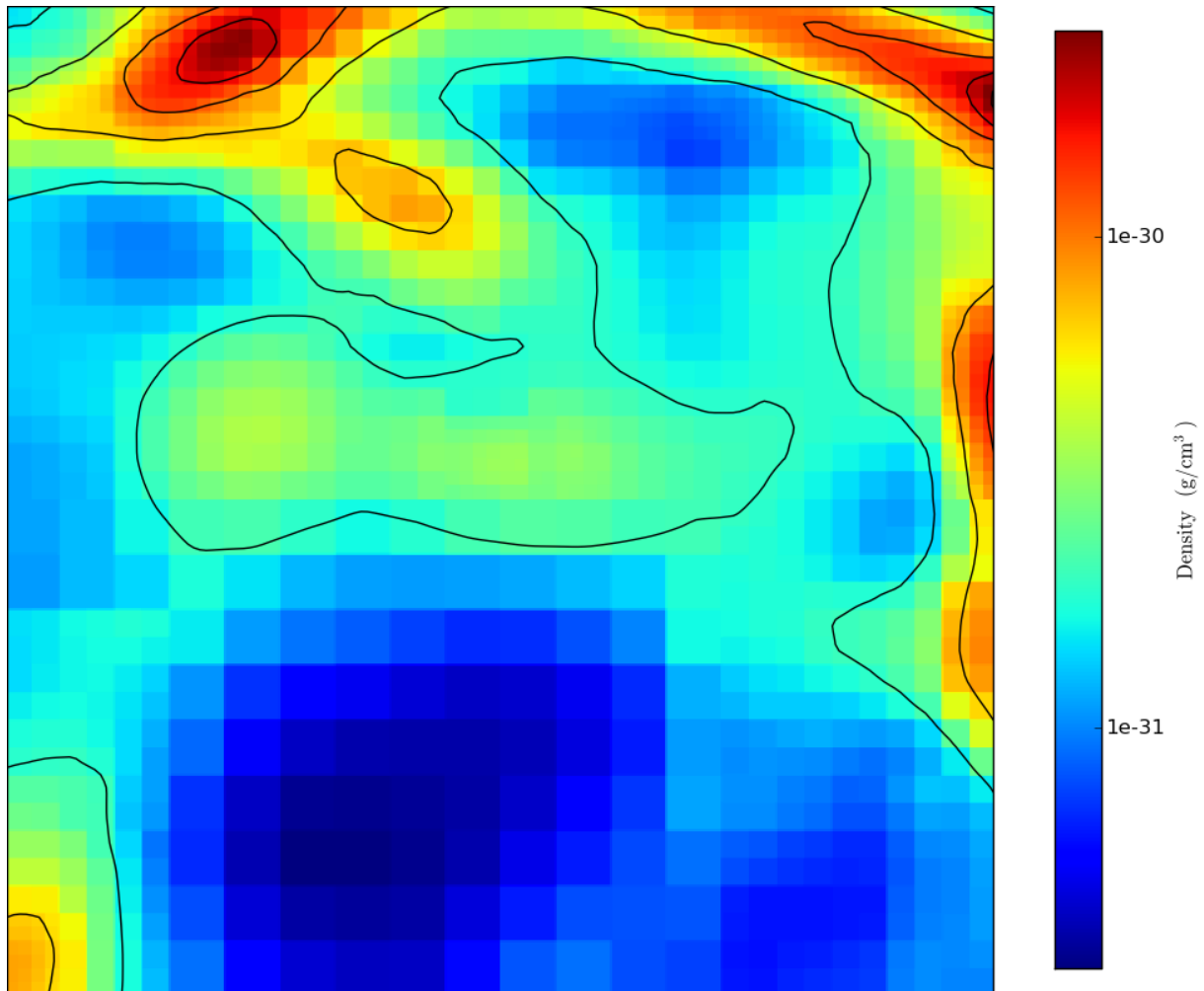
```

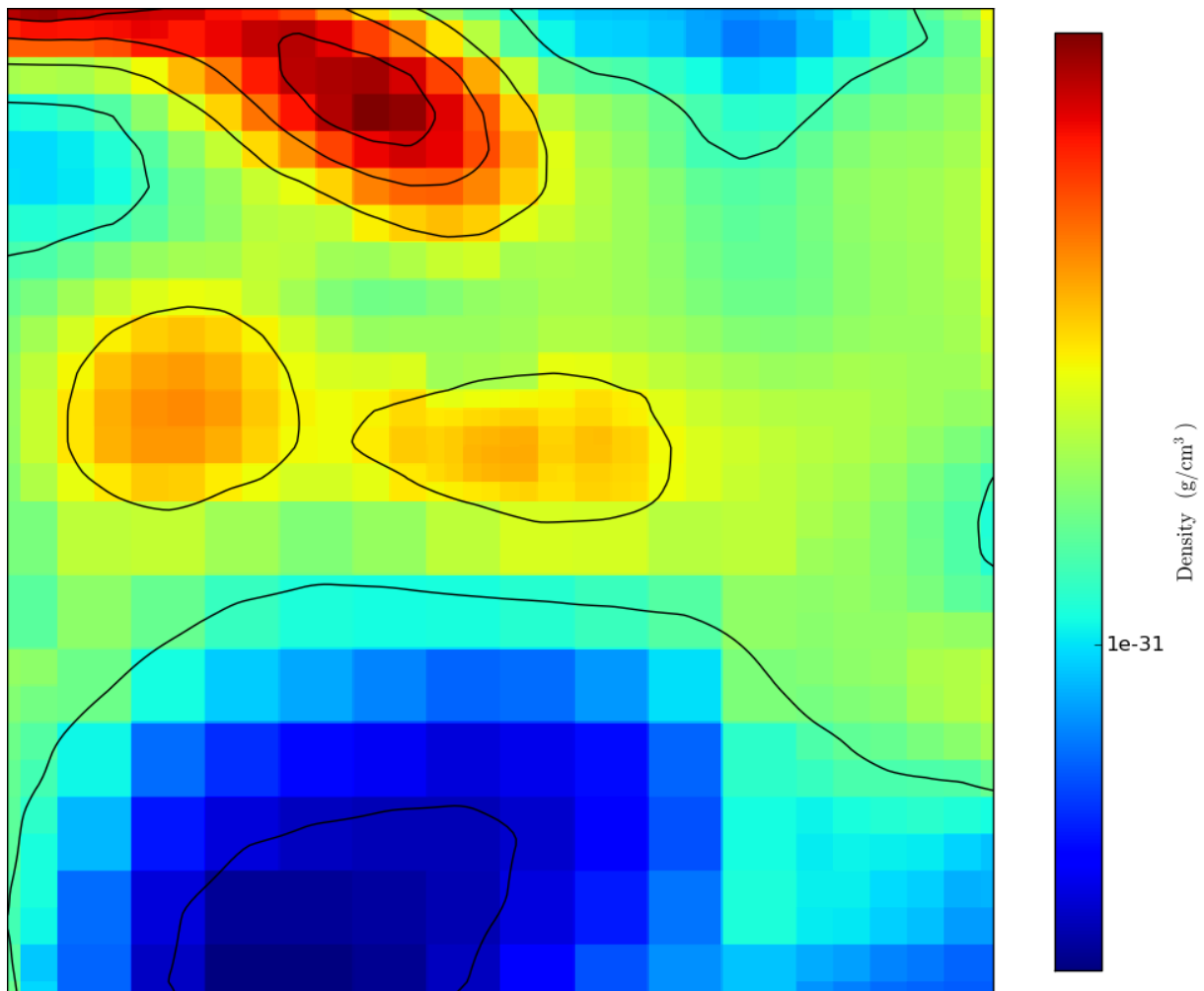
## Sample Output

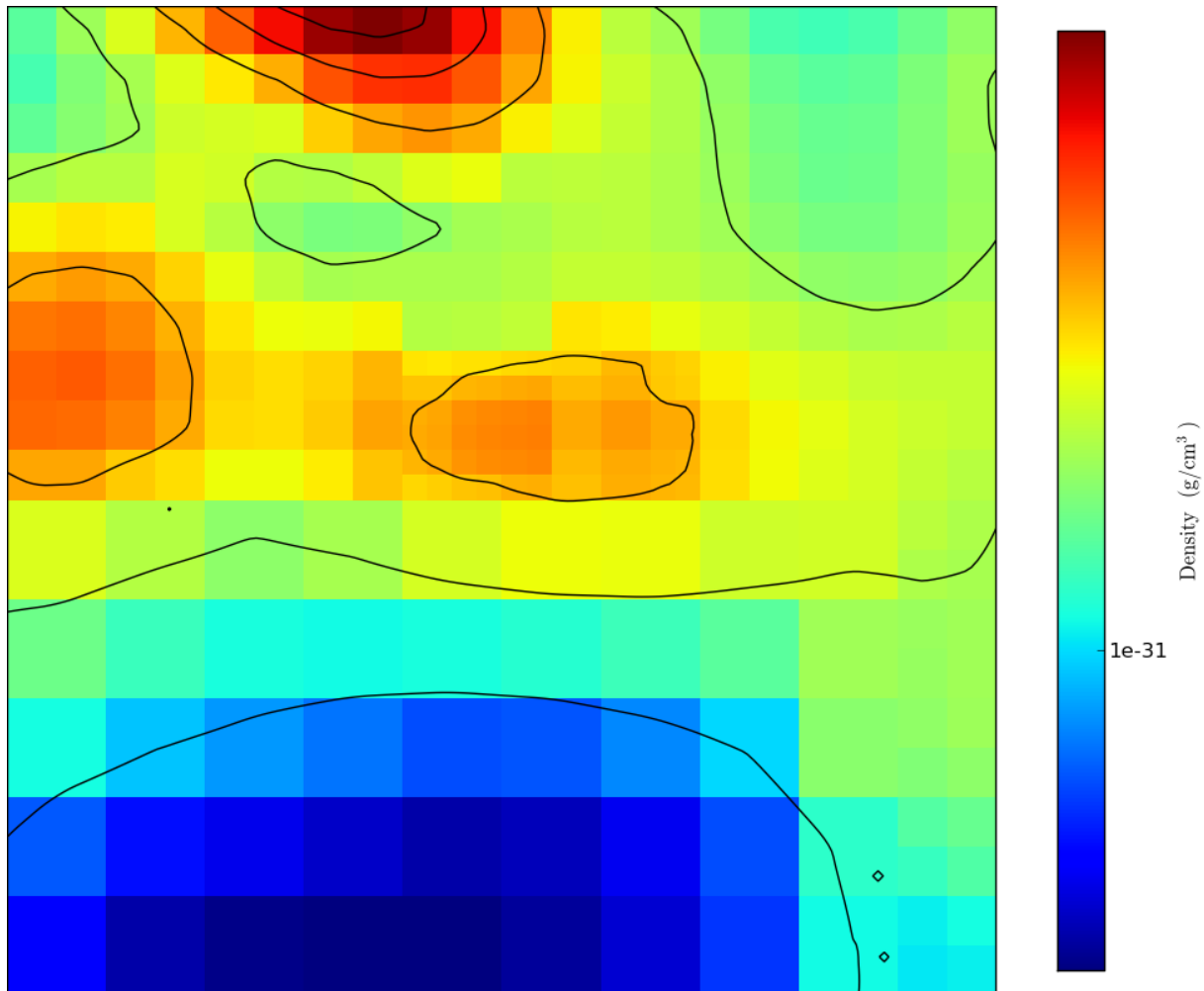












## 5.18 Overplot particles

This is a simple recipe to show how to open a dataset, plot a projection through it, and add particles on top

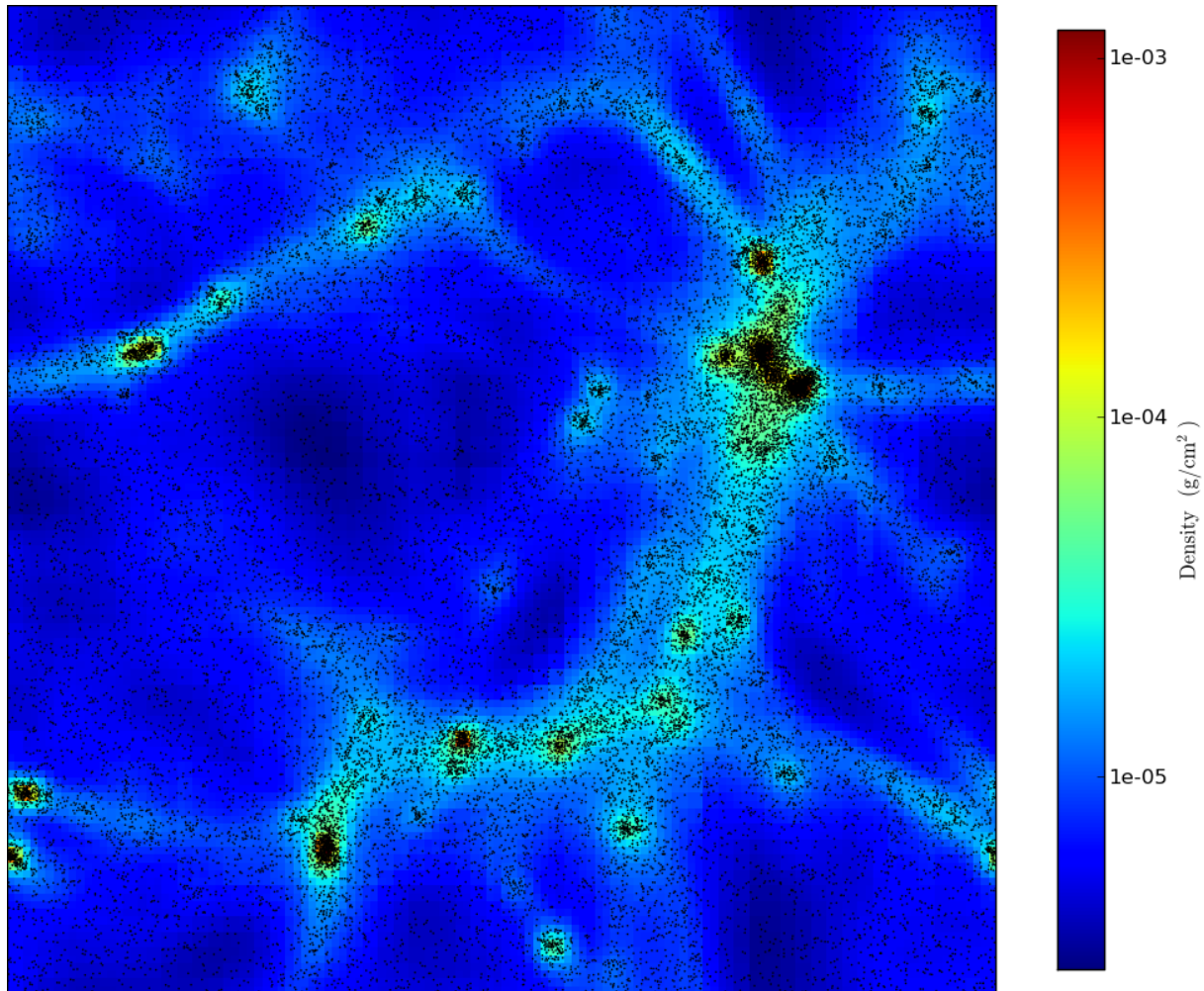
The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/overplot\\_particles.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/overplot_particles.py).

```
from yt.mods import * # set up our namespace

fn = "RedshiftOutput0005" # parameter file to load

pf = load(fn) # load data
pc = PlotCollection(pf, center=[0.5,0.5,0.5]) # defaults to center at most dense point
p = pc.add_projection("Density", 0) # 0 = x-axis
# "nparticles" is slightly more efficient than "particles"
p.modify["nparticles"](1.0) # 1.0 is the 'width' we want for our slab of
                             # particles -- this governs the allowable locations
                             # of particles that show up on the image
                             # NOTE: we can also supply a *ptype* to cut based
                             # on a given (integer) particle type
pc.set_width(1.0, '1') # change width of our plot to the full domain
pc.save(fn) # save all plots
```

## Sample Output



## 5.19 Multi plot

This is a simple recipe to show how to open a dataset and then plot a slice through it, centered at its most dense point.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/multi\\_plot.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/multi_plot.py).

```
from yt.mods import * # set up our namespace
import matplotlib.colorbar as cb

fn = "RedshiftOutput0005" # parameter file to load
orient = 'horizontal'

pf = load(fn) # load data

# There's a lot in here:
# From this we get a containing figure, a list-of-lists of axes into which we
# can place plots, and some axes that we'll put colorbars.
# We feed it:
# Number of plots on the x-axis, number of plots on the y-axis, and how we
```

---

```

# want our colorbars oriented. (This governs where they will go, too.
# bw is the base-width in inches, but 4 is about right for most cases.
fig, axes, colorbars = raven.get_multi_plot( 2, 1, colorbar=orient, bw = 4)

# We'll use a plot collection, just for convenience's sake
pc = PlotCollection(pf, center=[0.5, 0.5, 0.5])

# Now we add a slice and set the colormap of that slice, but note that we're
# feeding it an axes -- the zeroth row, the zeroth column, and telling the plot
# "Don't make a colorbar." We'll make one ourselves.
p = pc.add_slice("Density", 0, figure = fig, axes = axes[0][0], use_colorbar=False)
p.set_cmap("bds_highcontrast") # this is our colormap

# We do this again, but this time we take the 1-index column.
p = pc.add_slice("Temperature", 0, figure=fig, axes=axes[0][1], use_colorbar=False)
p.set_cmap("hot") # a different colormap

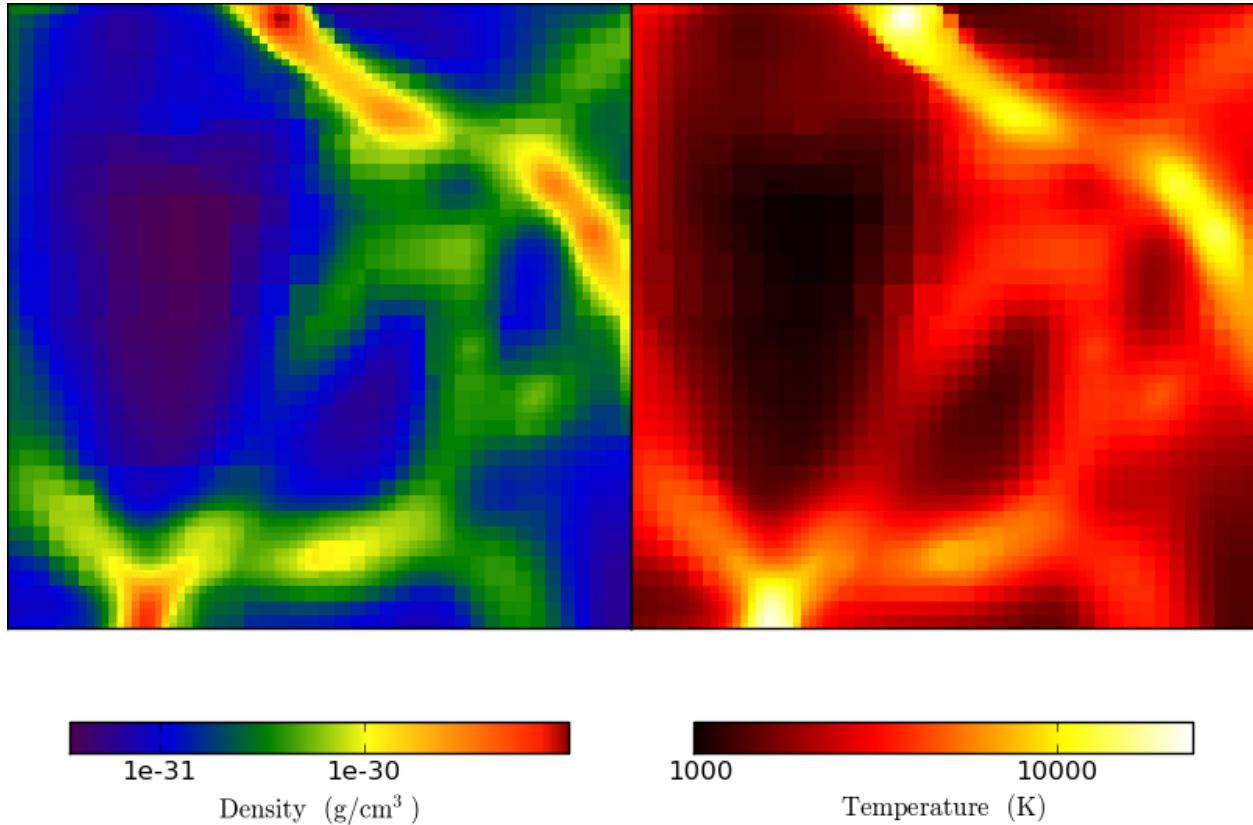
pc.set_width(5.0, 'mpc') # change width of both plots

# Each 'p' is a plot -- this is the Density plot and the Temperature plot.
# Each 'cax' is a colorbar-container, into which we'll put a colorbar.
# zip means, give these two me together.
for p, cax in zip(pc.plots, colorbars):
    # Now we make a colorbar, using the 'image' attribute of the plot.
    # 'image' is usually not accessed; we're making a special exception here,
    # though. 'image' will tell the colorbar what the limits of the data are.
    cbar = cb.Colorbar(cax, p.image, orientation=orient)
    # Now, we have to do a tiny bit of magic -- we tell the plot what its
    # colorbar is, and then we tell the plot to set the label of that colorbar.
    p.colorbar = cbar
    p._autoset_label()

# And now we're done! Note that we're calling a method of the figure, not the
# PlotCollection.
fig.savefig("%s" % pf)

```

## Sample Output



## 5.20 Multi plot 3x2

This is a simple recipe to show how to open a dataset and then plot a slice through it, centered at its most dense point.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/multi\\_plot\\_3x2.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/multi_plot_3x2.py).

```
from yt.mods import * # set up our namespace
import matplotlib.colorbar as cb

fn = "RedshiftOutput0005" # parameter file to load
orient = 'horizontal'

pf = load(fn) # load data

# There's a lot in here:
# From this we get a containing figure, a list-of-lists of axes into which we
# can place plots, and some axes that we'll put colorbars.
# We feed it:
# Number of plots on the x-axis, number of plots on the y-axis, and how we
# want our colorbars oriented. (This governs where they will go, too.
# bw is the base-width in inches, but 4 is about right for most cases.
fig, axes, colorbars = raven.get_multi_plot( 2, 3, colorbar=orient, bw = 4)
```

```

# We'll use a plot collection, just for convenience's sake
pc = PlotCollection(pf, center=[0.5, 0.5, 0.5])

# Now we follow the method of "multi_plot.py" but we're going to iterate
# over the columns, which will become axes of slicing.
for ax in range(3):
    p = pc.add_slice("Density", ax, figure = fig, axes = axes[ax][0],
                    use_colorbar=False)
    p.set_cmap("bds_highcontrast") # this is our colormap
    p.set_zlim(5e-32, 1e-29)
    # We do this again, but this time we take the 1-index column.
    p = pc.add_slice("Temperature", ax, figure=fig, axes=axes[ax][1],
                    use_colorbar=False)
    p.set_zlim(1e3, 3e4) # Set this so it's the same for all.
    p.set_cmap("hot") # a different colormap

pc.set_width(5.0, 'mpc') # change width of both plots

# Each 'p' is a plot -- this is the Density plot and the Temperature plot.
# Each 'cax' is a colorbar-container, into which we'll put a colorbar.
# zip means, give these two me together. Note that it cuts off after the
# shortest iterator is exhausted, in this case pc.plots.
for p, cax in zip(pc.plots, colorbars):
    # Now we make a colorbar, using the 'image' attribute of the plot.
    # 'image' is usually not accessed; we're making a special exception here,
    # though. 'image' will tell the colorbar what the limits of the data are.
    cbar = cb.Colorbar(cax, p.image, orientation=orient)
    # Now, we have to do a tiny bit of magic -- we tell the plot what its
    # colorbar is, and then we tell the plot to set the label of that colorbar.
    p.colorbar = cbar
    p._autoset_label()

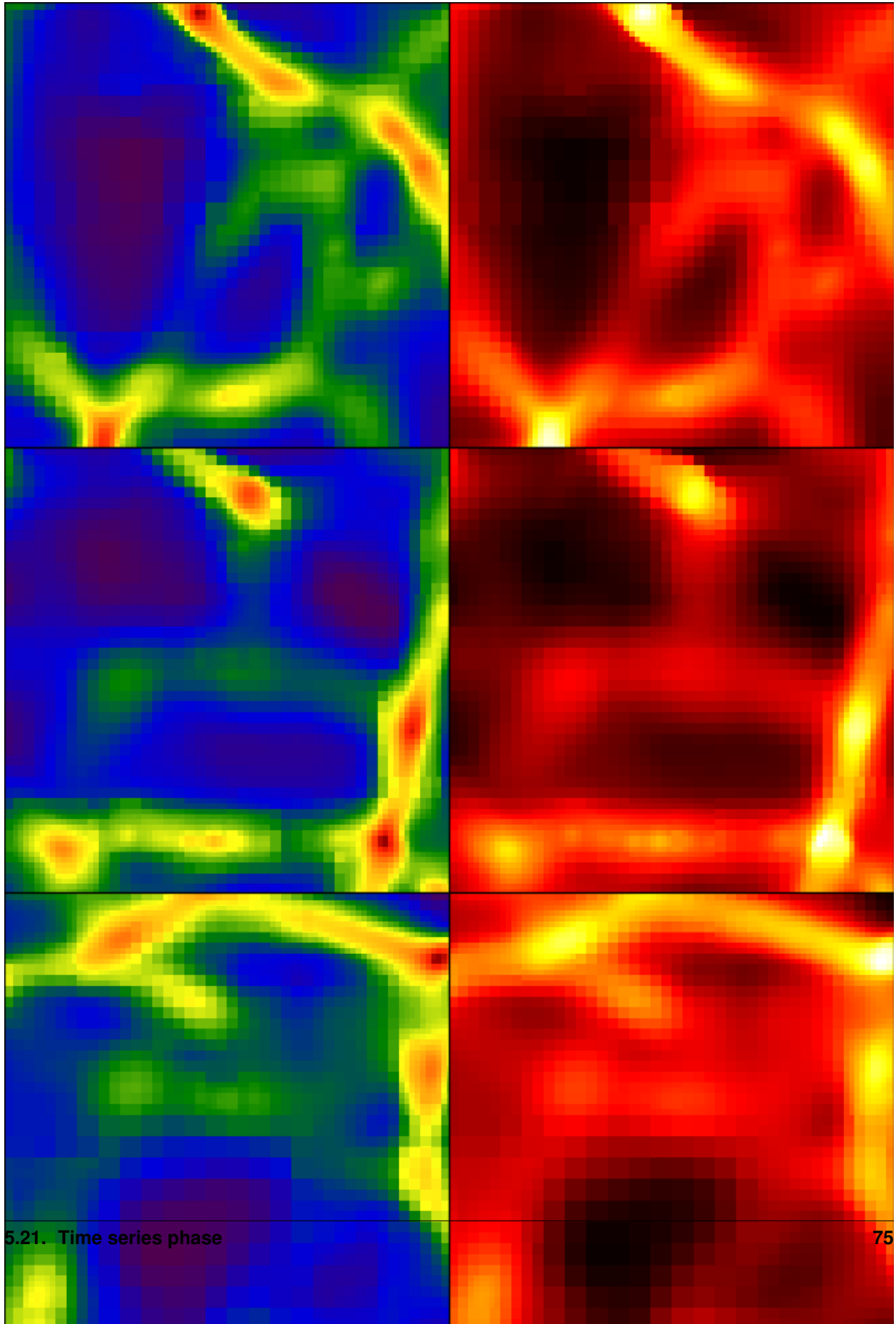
# And now we're done! Note that we're calling a method of the figure, not the
# PlotCollection.
fig.savefig("%s_3x2" % pf)

```





## Sample Output



If run with `mpirun` and the `-parallel` flag, this will take advantage of multiple processors.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/time\\_series\\_phase.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/time_series_phase.py).

```
from yt.mods import * # set up our namespace

# this means get all the parameter files that it can autodetect and then supply
# them as parameter file objects to the loop.
for pf in all_pfs(max_depth=2):
    # We create a plot collection to hold our plot
    # If we don't specify the center, it will look for one -- but we don't
    # really care where it's centered for this plot.
    pc = PlotCollection(pf, center=[0.5, 0.5, 0.5])

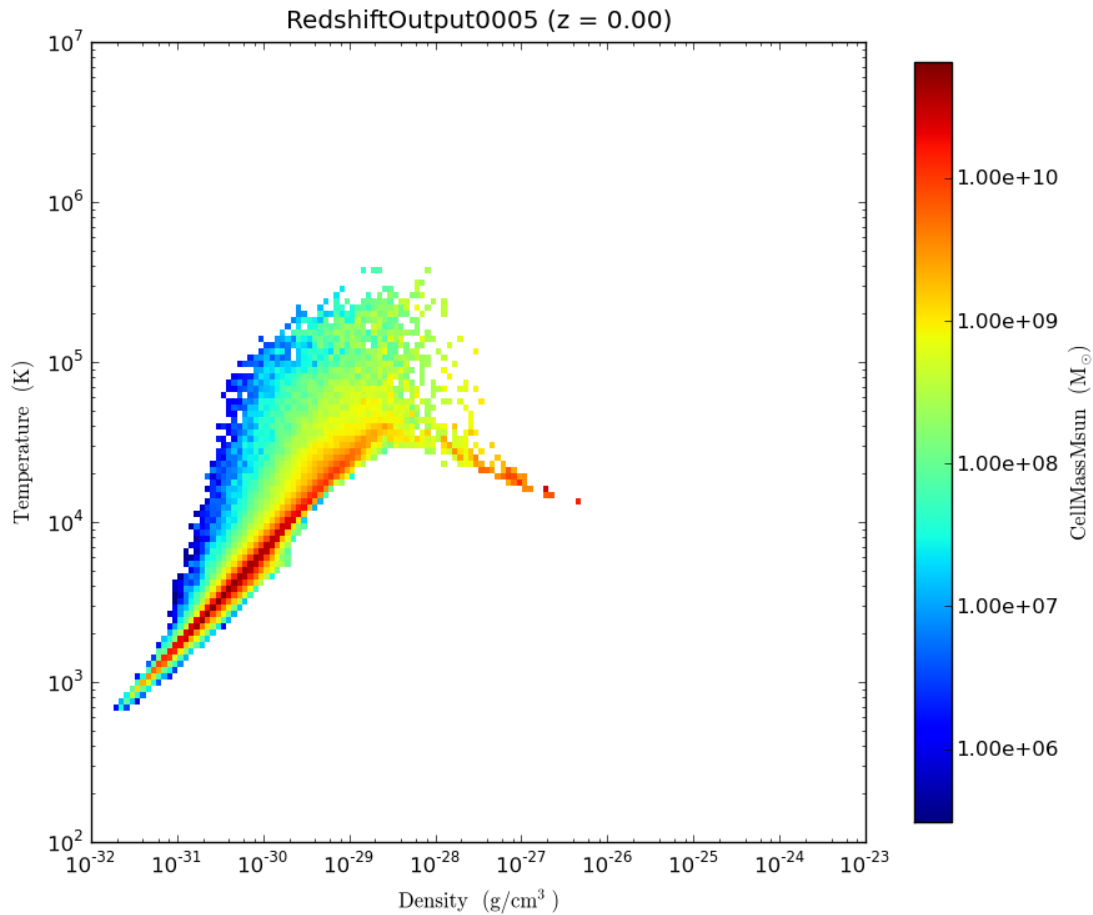
    # Now we add a phase plot of a sphere with radius 1.0 in code units.
    # If your domain is not 0..1, then this may not cover it completely.
    p = pc.add_phase_sphere(1.0, '1', ["Density", "Temperature", "CellMassMsun"],
                            weight=None, lazy_reader=True,
                            x_bins=128, x_bounds = (1e-32, 1e-24),
                            y_bins=128, y_bounds = (1e2, 1e7))

    # We've over-specified things -- but this will help ensure we have constant
    # bounds. lazy_reader gives it the go-ahead to run in parallel, and we
    # have asked for 128 bins from 1e-32 .. 1e-24 in Density-space and 128 bins
    # between 1e2 and 1e7 in Temperature space. This will lead to very fine
    # points of much lower mass, which is okay. You can reduce the number of
    # bins to get more mass in each bin. Additionally, weight=None means that
    # no averaging is done -- it just gets summed up, so the value of each bin
    # will be all the mass residing within that bin.

    # Now let's add a title with some fun information. p is the plot we were
    # handed previously. We will add the name of the parameter file and the
    # current redshift.
    p.modify["title"]("%s (z = %0.2f)" % (pf, pf["CosmologyCurrentRedshift"]))

    # Now let's save it out.
    pc.save() # "%s" % pf)
```

## Sample Output



## 5.22 Time series quantity

This is a recipe to sit inside a directory and calculate a quantity for all of the outputs in that directory.

If run with `mpirun` and the `-parallel` flag, this will take advantage of multiple processors.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/time\\_series\\_quantity.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/time_series_quantity.py).

```
from yt.mods import * # set up our namespace

# First set up our times and quantities lists
times = []
values = []

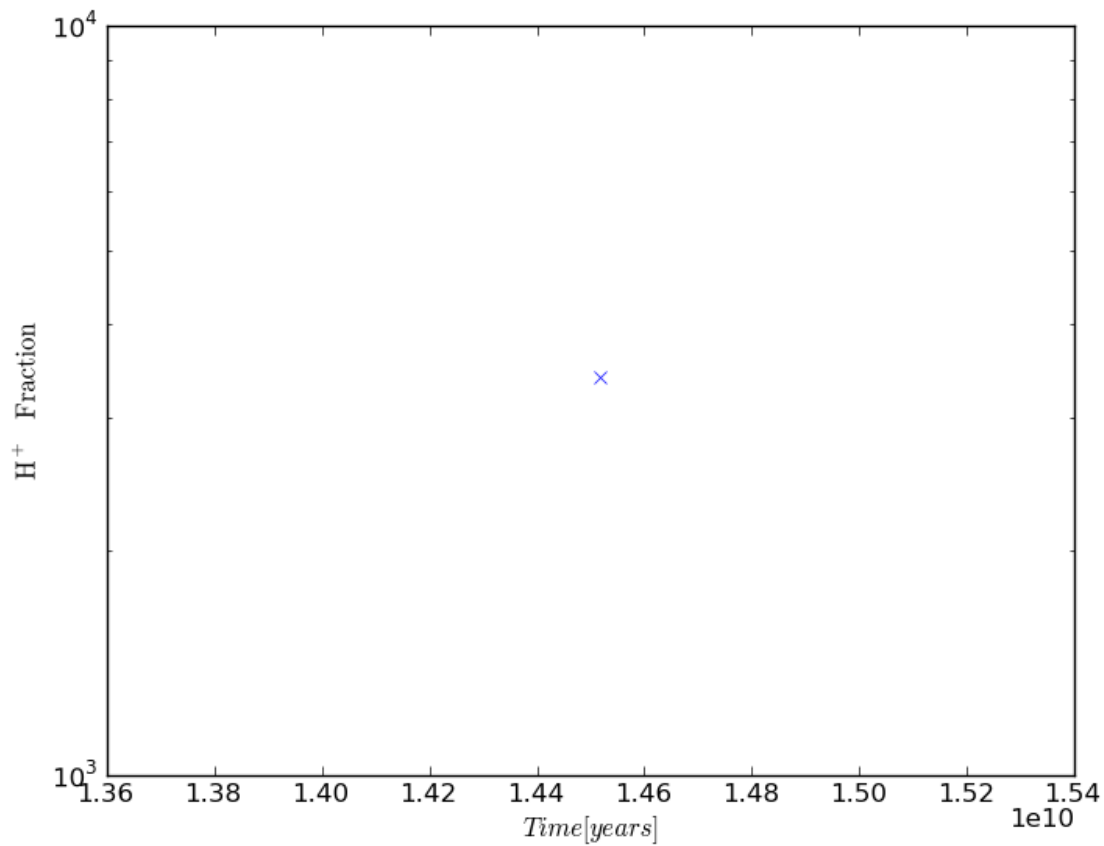
# this means get all the parameter files that it can autodetect and then supply
# them as parameter file objects to the loop.
for pf in all_pfs(max_depth=2):
    # Get the current time, convert to years from code units
    times.append(pf["InitialTime"] * pf["years"])
```

```
# Now get a box containing the entire dataset
data = pf.h.all_data()
# Now we calculate the average. The first argument is the quantity to
# average, the second is the weight.
# "lazy_reader" has two meanings -- the first is that it will try to
# operate on each individual grid, rather than a flattened array of all the
# data. The second is that it will also distribute grids across multiple
# processors, if multiple processors are in use.
val = data.quantities["WeightedAverageQuantity"](
    "Temperature", "CellVolume", lazy_reader=True)
values.append(val)

# Now we have our values and our time. We can plot this in pylab!

import pylab
pylab.semilogy(times, values, '-x')
pylab.xlabel(r"$Time [years]$")
pylab.ylabel(r"$\mathrm{H}^{+} \backslash \backslash \backslash \mathrm{Fraction}$")
pylab.savefig("average_HII_fraction.png")
```

## Sample Output



## 5.23 Extract fixed resolution data

This is a recipe to show how to open a dataset and extract it to a file at a fixed resolution with no interpolation or smoothing. Additionally, this recipe shows how to insert a dataset into an external HDF5 file using h5py.

The latest version of this recipe can be downloaded here: [http://hg.enzotools.org/cookbook/raw-file/tip/recipes/extract\\_fixed\\_resolution\\_data.py](http://hg.enzotools.org/cookbook/raw-file/tip/recipes/extract_fixed_resolution_data.py).

```
from yt.mods import *

# For this example we will use h5py to write to our output file.
import h5py

fn = "RedshiftOutput0005" # parameter file to load
pf = load(fn) # load data

# This is the resolution we will extract at
DIMS = 128

# Now, we construct an object that describes the data region and structure we
# want
cube = pf.h.covering_grid(2, # The level we are willing to extract to; higher
```

```
        # levels than this will not contribute to the data!
    # Now we set our spatial extent...
    left_edge=[0.0, 0.0, 0.0],
    right_edge=[1.0, 1.0, 1.0],
    # How many dimensions along each axis
    dims=[DIMS,DIMS,DIMS],
    # And any fields to preload (this is optional!)
    fields=["Density"])

# Now we open our output file using h5py
# Note that we open with 'w' which will overwrite existing files!
f = h5py.File("my_data.h5", "w")

# We create a dataset at the root node, calling it density...
f.create_dataset("/density", data=cube["Density"])

# We close our file
f.close()
```

# ADVANCED YT USAGE

yt has been designed to be flexible, with several entry points.

## 6.1 Derived Quantities

Derived quantities are a way of operating on a collection of cells and returning a set of values that is fewer in number than the number of cells – for instance yt already knows about several.

### 6.1.1 Using Derived Quantities

Every 3D data object (see *Using and Manipulating Objects and Fields* and *Object Methodology*) provides a mechanism for access to derived quantities. These can be accessed via the `quantities` interface, like so:

```
pf = load("my_data")
dd = pf.h.all_data()
dd.quantities["AngularMomentumVector"]()
```

The following quantities are available via the `quantities` interface.

**AngularMomentumVector()** : ()

(This is a proxy for `yt.lagos._AngularMomentumVector()`.) This function returns the mass-weighted average angular momentum vector.

**BaryonSpinParameter()** : ()

(This is a proxy for `yt.lagos._BaryonSpinParameter()`.) This function returns the spin parameter for the baryons, but it uses the particles in calculating enclosed mass.

**BulkVelocity()** : ()

(This is a proxy for `yt.lagos._BulkVelocity()`.) This function returns the mass-weighted average velocity in the object.

**CenterOfMass()** : ()

(This is a proxy for `yt.lagos._CenterOfMass()`.) This function takes no arguments and returns the location of the center of mass of the *non-particle* data in the object.

**Extrema(fields)** : ()

(This is a proxy for `yt.lagos._Extrema()`.) This function returns the extrema of a set of fields :param fields: A field name, or a list of field names

**IsBound(truncate=True, include\_thermal\_energy=False)** : ()

(This is a proxy for `yt.lagos._IsBound()`.) This returns whether or not the object is gravitationally bound. :param truncate: Should the calculation stop once the ratio of gravitational:kinetic is 1.0? :param

`include_thermal_energy`: Should we add the energy from `ThermalEnergy` on to the kinetic energy to calculate binding energy?

**MaxLocation(field):()**

(This is a proxy for `yt.lagos._MaxLocation()`.) This function returns the location of the maximum of a set of fields.

**ParticleSpinParameter():()**

(This is a proxy for `yt.lagos._ParticleSpinParameter()`.) This function returns the spin parameter for the baryons, but it uses the particles in calculating enclosed mass.

**TotalMass():()**

(This is a proxy for `yt.lagos._TotalMass()`.) This function takes no arguments and returns the sum of cell masses and particle masses in the object.

**TotalQuantity(fields):()**

(This is a proxy for `yt.lagos._TotalQuantity()`.) This function sums up a given field over the entire region. :param fields: The fields to sum up

**WeightedAverageQuantity(field, weight):()**

(This is a proxy for `yt.lagos._WeightedAverageQuantity()`.) This function returns an averaged quantity. :param field: The field to average :param weight: The field to weight by

## 6.1.2 Creating Derived Quantities

The basic idea is that you need to be able to operate both on a set of data, and a set of sets of data. (If this is not possible, the quantity needs to be added with the `force_unlazy` option.)

Two functions are necessary. One will operate on arrays of data, either fed from each grid individually or fed from the entire data object at once. The second one takes the results of the first, either as lists of arrays or as single arrays, and returns the final values. For an example, we look at the `TotalMass` function:

```
def _TotalMass(data):
    baryon_mass = data["CellMassMsun"].sum()
    particle_mass = data["ParticleMassMsun"].sum()
    return baryon_mass, particle_mass
def _combTotalMass(data, baryon_mass, particle_mass):
    return baryon_mass.sum() + particle_mass.sum()
add_quantity("TotalMass", function=_TotalMass,
             combine_function=_combTotalMass, n_ret = 2)
```

Once the two functions have been defined, we then call `add_quantity()` to tell it the function that defines the data, the collator function, and the number of values that get passed between them. In this case we return both the particle and the baryon mass, so we have two total values passed from the main function into the collator.

## 6.2 Plot Modification Mechanisms

Because the plots in `yt` are considered to be “volatile” – existing independent of the canvas on which they are plotted – before they are saved, you can have a set of “callbacks” run that modify them before saving to disk. By adding a callback, you are telling the plot that whatever it does itself, your callback gets the last word.

These can all be accessed through a registry attached to every plot object. When you add a plot to a `yt.raven.PlotCollection`, you get back that affiliated plot object. By accessing `modify` on that plot object, you have access to the available callbacks. For instance,



```
p = PlotCollection.add_slice("Density", 0)
p.modify["velocity"]()
```

would add the `velocity()` callback to the plot object. When you save the plot, the list of callbacks will be iterated over, and the velocity callback will be handed the current state of the plot. It will then be able to dynamically modify the plot before saving – in this case, adding on velocity vectors atop the image.

## 6.2.1 Available Callbacks

These are the callbacks available through the `modify[]` mechanism. The underlying functions are documented (largely identical to this) in *yt.raven.FixedResolution Pixelization Interface*.

**arrow(self, pos, code\_size, plot\_args=None):()**

(This is a proxy for `yt.raven.Callbacks.ArrowCallback`.) This adds an arrow pointing at *pos* with size *code\_size* in code units. *plot\_args* is a dict fed to matplotlib with arrow properties.

**clumps(self, clumps, plot\_args=None):()**

(This is a proxy for `yt.raven.Callbacks.ClumpContourCallback`.) Take a list of *clumps* and plot them as a set of contours.

**contour(self, field, ncont=5, factor=4, take\_log=False, clim=None, plot\_args=None):()**

(This is a proxy for `yt.raven.Callbacks.ContourCallback`.) Add contours in *field* to the plot. *ncont* governs the number of contours generated, *factor* governs the number of points used in the interpolation, *take\_log* governs how it is contoured and *clim* gives the (upper, lower) limits for contouring.

**coord\_axes(self, unit=None, coords=False):()**

(This is a proxy for `yt.raven.Callbacks.CoordAxesCallback`.) Creates x and y axes for a VMPlot. In the future, it will attempt to guess the proper units to use.

**quiver(self, field\_x, field\_y, factor):()**

(This is a proxy for `yt.raven.Callbacks.CuttingQuiverCallback`.) Get a quiver plot on top of a cutting plane, using *field\_x* and *field\_y*, skipping every *factor* datapoint in the discretization.

**grids(self, alpha=1.0, min\_pix=1):()**

(This is a proxy for `yt.raven.Callbacks.GridBoundaryCallback`.) Adds grid boundaries to a plot, optionally with *alpha*-blending. Cutoff for display is at *min\_pix* wide.

**hop\_circles(self, hop\_output, max\_number=None, annotate=False, min\_size=20, max\_size=100000):()**

(This is a proxy for `yt.raven.Callbacks.HopCircleCallback`.) Accepts a `yt.lagos.HopList` *hop\_output* and plots up to *max\_number* (None for unlimited) halos as circles.

**hop\_particles(self, hop\_output, p\_size=1.0, max\_number=None, min\_size=20, alpha=0.20000000):()**

(This is a proxy for `yt.raven.Callbacks.HopParticleCallback`.) Adds particle positions for the members of each halo as identified by HOP. Along *axis* up to *max\_number* groups in *hop\_output* that are larger than *min\_size* are plotted with *p\_size* pixels per particle; *alpha* determines the opacity of each particle.

**axis\_label(self, label):()**

(This is a proxy for `yt.raven.Callbacks.LabelCallback`.) This adds a label to the plot.

**line(self, x, y, plot\_args=None):()**

(This is a proxy for `yt.raven.Callbacks.LinePlotCallback`.) Over plot *x* and *y* with *plot\_args* fed into the plot.

**marker(self, pos, marker='x', plot\_args=None):()**

(This is a proxy for `yt.raven.Callbacks.MarkerAnnotateCallback`.) Adds text *marker* at *pos* in code-arguments. *plot\_args* is a dict that will be forwarded to the plot command.

**nparticles**(self, width, p\_size=1.0, col='k', stride=1.0, ptype=None):()  
(This is a proxy for `yt.raven.Callbacks.NewParticleCallback`.) Adds particle positions, based on a thick slab along *axis* with a *width* along the line of sight. *p\_size* controls the number of pixels per particle, and *col* governs the color. *ptype* will restrict plotted particles to only those that are of a given type.

**particles**(self, axis, width, p\_size=1.0, col='k', stride=1.0):()  
(This is a proxy for `yt.raven.Callbacks.ParticleCallback`.) Adds particle positions, based on a thick slab along *axis* with a *width* along the line of sight. *p\_size* controls the number of pixels per particle, and *col* governs the color.

**point**(self, pos, text, text\_args=None):()  
(This is a proxy for `yt.raven.Callbacks.PointAnnotateCallback`.) This adds *text* at position *pos*, where *pos* is in code-space. *text\_args* is a dict fed to the text placement code.

**quiver**(self, field\_x, field\_y, factor):()  
(This is a proxy for `yt.raven.Callbacks.QuiverCallback`.) Adds a ‘quiver’ plot to any plot, using the *field\_x* and *field\_y* from the associated data, skipping every *factor* datapoints.

**sphere**(self, center, radius, circle\_args=None, text=None, text\_args=None):()  
(This is a proxy for `yt.raven.Callbacks.SphereCallback`.) A sphere centered at *center* in code units with radius *radius* in code units will be created, with optional *circle\_args*, *text*, and *text\_args*.

**text**(self, pos, text, text\_args=None):()  
(This is a proxy for `yt.raven.Callbacks.TextLabelCallback`.) Accepts a position in (0..1, 0..1) of the image, some text and optionally some text arguments.

**title**(self, title='Plot'):()  
(This is a proxy for `yt.raven.Callbacks.TitleCallback`.) Accepts a *title* and adds it to the plot

**units**(self, unit='au', factor=4, text\_annotate=True, text\_which=-2):()  
(This is a proxy for `yt.raven.Callbacks.UnitBoundaryCallback`.) Add on a plot indicating where *factor*\**s of unit* are shown. Optionally *text\_annotate* on the *text\_which*-indexed box on display.

**velocity**(self, factor=16):()  
(This is a proxy for `yt.raven.Callbacks.VelocityCallback`.) Adds a ‘quiver’ plot of velocity to the plot, skipping all but every *factor* datapoint

## 6.3 The Plugin File

The plugin file is a means of modifying the available fields, quantities, data objects and so on without modifying the source code of yt.

The following configuration parameters need to be set in the `~/yt/config` file in order to enable the usage of a plugin file:

```
[lagos]

loadfieldplugins: True
pluginfilename: my_plugins.py
```

You can call your plugin file whatever you like, and after the imports inside the `lagos` module are completed, it is executed in that namespace.

The code in this file can thus add fields, add derived quantities, add datatypes, and on and on. For example, if I created a plugin file containing:

```
def _myfunc(field, data):
    return na.random.random(data["Density"].shape)
add_field("SomeQuantity", function=_myfunc)
```

then all of my data objects would have access to the field “SomeQuantity” despite its lack of use.

## 6.4 Creating 3D Datatypes

The three-dimensional datatypes in yt follow a fairly simple protocol. The basic principle is that if you want to define a region in space, that region must be identifiable from some sort of cut applied against the cells – typically, in yt, this is done by examining the geometry. (The `yt.lagos.ExtractedRegionBase` type is a notable exception to this, as it is defined as a subset of an existing data object.)

In principle, you can define any number of 3D data objects, as long as the following methods are implemented to protocol specifications.

**`__init__`** (*self, args, kwargs*)

This function can accept any number of arguments but must eventually call `AMR3DData.__init__`. It is used to set up the various parameters that define the object.

**`_get_list_of_grids`** (*self*)

This function must set the property `_grids` to be a list of the grids that should be considered to be a part of the data object. Each of these will be partly or completely contained within the object.

**`_is_fully_enclosed`** (*self, grid*)

This function returns true if the entire grid is part of the data object and false if it is only partly enclosed.

**`_get_cut_mask`** (*self, grid*)

This function returns a boolean mask in the shape of the grid. All of the cells set to ‘True’ will be included in the data object and all of those set to ‘False’ will be excluded. Typically this is done via some logical operation.

For a good example of how to do this, see the `yt.lagos.AMRCylinderBase` source code.

## 6.5 Debugging and Driving YT

There are several different convenience functions that allow you to control YT in perhaps unexpected and unorthodox manners. These will allow you to conduct in-depth debugging of processes that may be running in parallel on multiple processors, as well as providing a mechanism of signalling to YT that you need more information about a running process. Additionally, YT has a built-in mechanism for optional reporting of errors to a central server. All of these allow for more rapid development and debugging of any problems you might encounter.

Additionally, yt is able to leverage existing developments in the IPython community for parallel, interactive analysis. This allows you to initialize multiple YT processes through `mpirun` and interact with all of them from a single, unified interactive prompt. This enables and facilitates parallel analysis without sacrificing interactivity and flexibility.

### 6.5.1 The Pastebin

At <http://paste.enzotools.org/> a pastebin is available for placing scripts. With yt the script `yt_lodgeit.py` is distributed, which allows for commandline uploading and downloading of pasted snippets. To upload script you would supply it to the command:

```
$ yt_lodgeit.py some_script.py
```

The URL will be returned. If you'd like it to be marked 'private' and not show up in the list of pasted snippets, supply the argument `--private`. All snippets are given either numbers or hashes. To download a pasted snippet, you would use the `--download` option:

```
$ yt_lodgeit.py --download=216
```

The snippet will be output to the window, so output redirection can be used to store it in a file.

## Error Reporting with the Pastebin

If you are having troubles with `yt`, you can have it paste the error report to the pastebin by running your problematic script with the `--paste` option:

```
$ python2.6 some_problematic_script.py --paste
```

The `--paste` option has to come after the name of the script. When the script dies and prints its error, it will also submit that error to the pastebin and return a URL for the error. When reporting your bug, include this URL and then the problem can be debugged more easily.

For more information on asking for help, see *asking-for-help*.

## 6.5.2 Signaling YT to Do Something

During startup, `yt` inserts handlers for two operating system-level signals. These provide two diagnostic methods for interacting with a running process. Signalling the python process that is running your script with these signals will induce the requested behavior.

**SIGUSR1** This will cause the python code to print a stack trace, showing exactly where in the function stack it is currently executing.

**SIGUSR2** This will cause the python code to throw a `RuntimeError`. If you are running with the `--rpdb` options (see *Remote and Disconnected Debugging*) this will also cause the interpreter to sit inside a debug loop. If you are running inside `pdb` (see `pdb`) then `pdb` will produce a debug loop.

If your `yt`-running process has PID 5829, you can signal it to print a traceback with:

```
$ kill -SIGUSR1 5829
```

Note, however, that if the code is currently inside a C function, the signal will not be handled, and the stacktrace will not be printed, until it returns from that function.

## 6.5.3 Remote and Disconnected Debugging

If you are running a parallel job that fails, often it can be difficult to do a post-mortem analysis to determine what went wrong. To facilitate this, `yt` has implemented an **XML-RPC** interface to the Python debugger (`pdb`) event loop.

Running with the `--rpdb` command will cause any uncaught exception during execution to spawn this interface, which will sit and wait for commands, exposing the full Python debugger. Additionally, a frontend to this is provided through the `yt` command. So if you run the command:

```
$ mpirun -np 4 python2.6 some_script.py --parallel --rpdb
```

and it reaches an error or an exception, it will launch the debugger. Additionally, instructions will be printed for connecting to the debugger. Each of the four processes will be accessible via:

```
$ yt rpdb 0
```

where 0 here indicates the process 0.

For security reasons, this will only work on local processes; to connect on a cluster, you will have to execute the command `yt rpdb` on the node on which that process was launched.

## 6.5.4 Interactive Parallel Processing with IPython

IPython is a powerful mechanism not only for interactive usage, but also for task delegation and parallel analysis driving. Using the [IPython Parallel Multi Engine](#) interface, you can launch multiple ‘engines’ for computation which can then be driven by `yt`. However, to do so, you will have to ensure that the IPython dependencies for parallel computation are met – this requires the installation of a few components.

- [PyOpenSSL](#)
- [Twisted](#)
- [Foolscap](#)

Both Twisted and Foolscap can be installed using `easy_install` but PyOpenSSL requires manual installation. Of course, `yt` itself requires [mpi4py](#) to be installed as well, which is described in [Parallel Computation With YT](#).

The entire section in the IPython manual on [parallel computation](#) is essential reading, but for a quick start, you need to launch the engines:

```
$ ipcontroller
$ mpirun -np 4 ipengine
```

This will launch the controller, which interfaces with the new computation engines launched afterward. Note that you can launch an arbitrary number of compute processes. Now, launch IPython:

```
$ ipython
```

and execute the commands:

```
ipcontroller mpirun -np 4 ipengine
mec = client.MultiEngineClient()
mec.activate()
```

You have now obtained an object, `mec`, which is able to interact with and control all of the launched engines. Any command prefixed with the string `%px` will now be issued on all processors. Any action that would be executed in parallel in `yt` will be executed in parallel here. For instance,

```
%px from yt.mods import *
%px pf = load("data0050")
%px pc = PlotCollection(pf)
%px pc.add_projection("Density", 0)
```

This will load up the name space, the parameter file, and project through `data0050` in parallel utilizing all of our processors. IPython can also execute commands on a limited subset of hosts, and it can also turn on auto-execution, to send all of your commands to all of the compute engines, using the `%autopx` directive.



# EXTENSIONS

Extensions take `yt` fundamentals and run with them. For certain analysis needs, these tools make life a lot easier.

## 7.1 Halo Finding

*Section author: Stephen Skory <[sskory@physics.ucsd.edu](mailto:sskory@physics.ucsd.edu)>*

There are two methods of finding particle haloes in `yt`. The recommended and default method is called HOP, a method described in [Eisenstein and Hut \(1998\)](#). A basic friends-of-friends (e.g. [Efstathiou et al. \(1985\)](#)) halo finder is also implemented, however at this time it should be considered experimental.

### 7.1.1 HOP

The version of HOP used in `yt` is an upgraded version of the [publicly available HOP code](#). Support for 64-bit floats and integers has been added, as well as parallel analysis through spatial decomposition. HOP builds groups in this fashion:

1. Estimates the local density at each particle using a smoothing kernel.
2. Builds 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 160.0.

Please see the [HOP method paper](#) for full details.

### 7.1.2 Friends-of-Friends

The version of FoF in `yt` is based on the [publicly available FoF code](#) from the University of Washington. Like HOP, FoF supports parallel analysis through spatial decomposition. FoF is much simpler than HOP:

1. From the total number of particles, and the volume of the region, the average inter-particle spacing is calculated.
2. Pairs of particles closer together than some fraction of the average inter-particle spacing (the default is 0.2) are linked together. Particles can be paired with more than one other particle.
3. The final groups are formed the networks of particles linked together by friends, hence the name.

**Warning:** The FoF halo finder in yt is not thoroughly tested! It is probably fine to use, but you are strongly encouraged to check your results against the data for errors.

### 7.1.3 Running HaloFinder

Running HOP on a dataset is straightforward

```
from yt.mods import *
pf = load("data0001")
halo_list = HaloFinder(pf)
:language: python
```

Running FoF is similar:

```
from yt.mods import *
pf = load("data0001")
halo_list = FOFHaloFinder(pf)
```

### 7.1.4 Halo Data Access

`halo_list` is a list of `Halo` class objects ordered by decreasing halo size. A `Halo` object has convenient ways to access halo data. This loop will print the location of the center of mass for each halo found

```
for halo in halo_list:
    print halo.center_of_mass()
```

All the methods are:

- `.center_of_mass()` - the center of mass for the halo.
- `.maximum_density()` - the maximum density in “HOP” units.
- `.maximum_density_location()` - the location of the maximum density particle in the HOP halo.
- `.total_mass()` - the mass of the halo in  $M_{\text{sol}}$  (not  $M_{\text{sol}}/h$ ).
- `.bulk_velocity()` - the velocity of the center of mass of the halo in simulation units.
- `.maximum_radius()` - the distance from the center of mass to the most distant particle in the halo in simulation units.
- `.get_size()` - the number of particles in the halo.
- `.get_sphere()` - returns an `EnzoSphere` object using the center of mass and maximum radius.

**Note:** For FOF the maximum density value is meaningless and is set to -1 by default. For FOF the maximum density location will be identical to the center of mass location.

The command

```
halo_list.write_out("HaloAnalysis.out")
```

will output the results of HOP or FoF to a text file named `HaloAnalysis.out`. The file contains each of the data values listed above except for `.get_sphere()`.

For each halo the data for the particles in the halo can be accessed like this

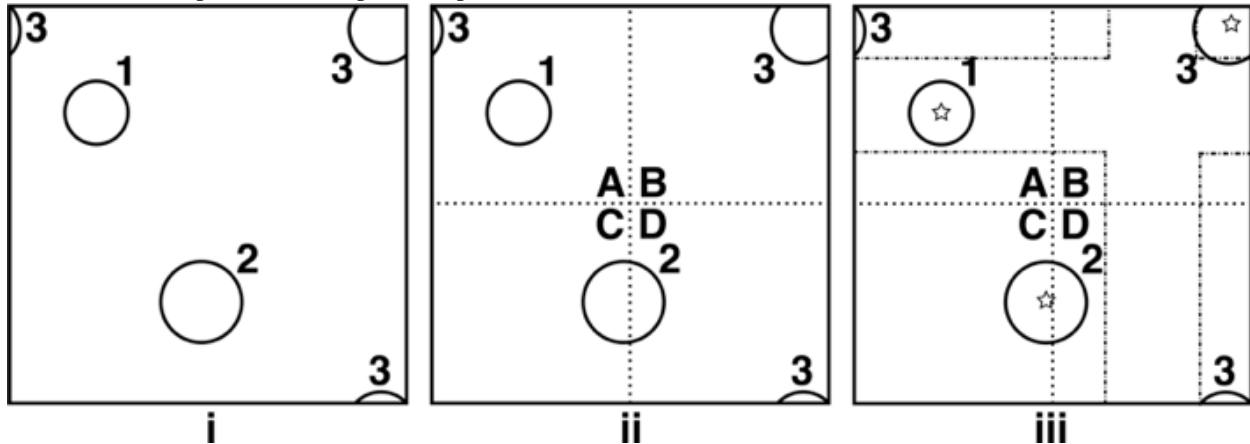


```
for halo in halo_list:
    print halo["particle_index"]
    print halo["particle_position_x"] # in simulation units
```

### 7.1.5 Parallel Halo Analysis

Both the HOP and FoF halo finders can run in parallel using spatial decomposition. In order to run them in parallel it is helpful to understand how it works.

Below in the first plot (i) is a simplified depiction of three haloes labeled 1,2 and 3:



Halo 3 is twice reflected around the periodic boundary conditions.

In (ii), the volume has been sub-divided into four equal subregions, A,B,C and D, shown with dotted lines. Notice that halo 2 is now in two different subregions, C and D, and that halo 3 is now in three, A, B and D. If the halo finder is run on these four separate subregions, halo 1 is identified as a single halo, but haloes 2 and 3 are split up into multiple haloes, which is incorrect. The solution is to give each subregion padding to oversample into neighboring regions.

In (iii), subregion C has oversampled into the other three regions, with the periodic boundary conditions taken into account, shown by dot-dashed lines. The other subregions oversample in a similar way.

The halo finder is then run on each padded subregion independently and simultaneously. By oversampling like this, haloes 2 and 3 will both be enclosed fully in at least one subregion and identified completely.

Haloes identified with centers of mass inside the padded part of a subregion are thrown out, eliminating the problem of halo duplication. The centers for the three haloes are shown with stars. Halo 1 will belong to subregion A, 2 to C and 3 to B.

#### Parallel HaloFinder padding

To run with parallel halo finding, there is a slight modification to the script

```
from yt.mods import *
pf = load("data0001")
halo_list = HaloFinder(pf, padding=0.02)
# --or--
halo_list = FOFHaloFinder(pf, padding=0.02)
```

The padding parameter is in simulation units and defaults to 0.02. This parameter is how much padding is added to each of the six sides of a subregion. This value should be 2x-3x larger than the largest expected halo in the simulation.

It is unlikely, of course, that the largest object in the simulation will be on a subregion boundary, but there is no way of knowing before the halo finder is run.

In general, a little bit of padding goes a long way, and too much just slows down the analysis and doesn't improve the answer (but doesn't change it). It may be worth your time to run the parallel halo finder at a few paddings to find the right amount, especially if you're analyzing many similar datasets.

## 7.2 HaloProfiler

*Section author: Britton Smith <[britton.smith@colorado.edu](mailto:britton.smith@colorado.edu)>*

The HaloProfiler provides a means of performing analysis on multiple points in a dataset at once. This is primarily intended for use with cosmological simulations, in which gravitationally bound structures composed of dark matter and gas, called halos, form and become the hosts for galaxies and galaxy clusters.

The HaloProfiler performs two primary functions: radial profiles and projections. With only a few exceptions discussed below, all of the HaloProfiler's machinery can be run in parallel, with `mpi4py` installed, by running your script inside an `mpirun` call with the `-parallel` flag at the end.

### 7.2.1 Configuring the HaloProfiler

A sample script to run the HaloProfiler can be found in `examples/run_halo_profiler.py`. In order to run the HaloProfiler on a dataset, a HaloProfiler object must be instantiated with the path to the dataset as the only argument:

```
import yt.extensions.HaloProfiler as HP
hp = HP.HaloProfiler("DD0242/DD0242")
```

Most of the HaloProfiler's options are configured with keyword arguments given at instantiation. These options are:

- **halos** (*str*): “multiple” for profiling more than one halo. In this mode halos are read in from a list or identified with a halo finder. In “single” mode, the one and only halo center is identified automatically as the location of the peak in the density field. Default: “multiple”.
- **halo\_list\_file** (*str*): name of file containing the list of halos. The HaloProfiler will look for this file in the data directory. Default: “HopAnalysis.out”.
- **halo\_list\_format** (*str* or *dict*): the format of the halo list file. “yt\_hop” for the format given by yt's halo finders. “enzo\_hop” for the format written by enzo\_hop. This keyword can also be given in the form of a dictionary specifying the column in which various properties can be found. For example, {“id”: 0, “center”: [1, 2, 3], “mass”: 4, “radius”: 5}. Default: “yt\_hop”.
- **halo\_finder\_function** (*function*): If halos is set to multiple and the file given by halo\_list\_file does not exist, the halo finding function specified here will be called. Default: HaloFinder(yt\_hop).
- **halo\_finder\_args** (*tuple*): args given with call to halo finder function. Default: None.
- **halo\_finder\_kwargs** (*dict*): kwargs given with call to halo finder function. Default: None.
- **use\_density\_center** (*bool*): re-center halos before performing profiles with an center of mass weighted by overdensity. This is generally not needed. Default: False.
- **density\_center\_exponent** (*flt*): when use\_density\_center set to True, this specifies the exponent, alpha, such that the halo center calculation is weighted by overdensity<sup>alpha</sup>. Default: 1.0.
- **use\_field\_max\_center** (*str*): another alternative for halo re-centering by selecting the location of the maximum of the field given by this keyword. This is generally not needed. Default: None.

- **halo\_radius** (*flt*): if no halo radii are provided in the halo list file, this parameter is used to specify the radius out to which radial profiles will be made. This keyword is also used when halos is set to single. Default: 0.1.
- **radius\_units** (*str*): the units of **halo\_radius**. Default: “1” (code units).
- **n\_profile\_bins** (*int*): the number of bins in the radial profiles. Default: 50.
- **profile\_output\_dir** (*str*): the subdirectory, inside the data directory, in which radial profile output files will be created. The directory will be created if it does not exist. Default: “radial\_profiles”.
- **projection\_output\_dir** (*str*): the subdirectory, inside the data directory, in which projection output files will be created. The directory will be created if it does not exist. Default: “projections”.
- **projection\_width** (*flt*): the width of halo projections. Default: 8.0.
- **projection\_width\_units** (*str*): the units of projection\_width. Default: “mpc”.
- **project\_at\_level** (*int* or “max”): the maximum refinement level to be included in projections. Default: “max” (maximum level within the dataset).
- **velocity\_center** (*list*): the method in which the halo bulk velocity is calculated (used for calculation of radial and tangential velocities. Valid options are:
  - [“bulk”, “halo”] (Default): the velocity provided in the halo list
  - [“bulk”, “sphere”]: the bulk velocity of the sphere centered on the halo center.
  - [“max”, field]: the velocity of the cell that is the location of the maximum of the field specified (used only when halos set to single).
- **filter\_quantities** (*list*): quantities from the original halo list file to be written out in the filtered list file. Default: [‘id’, ‘center’].

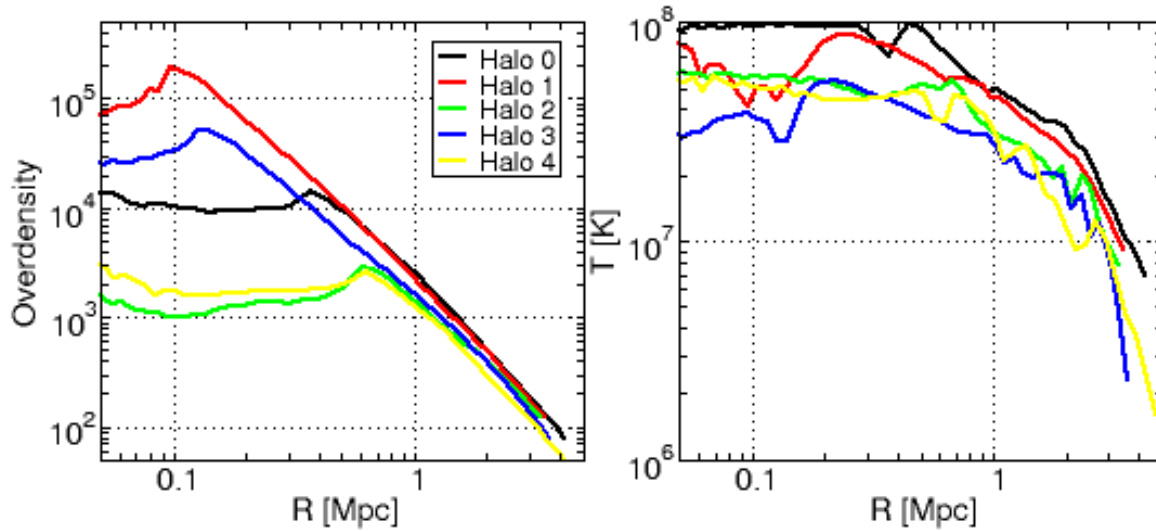
**Warning:** The HaloProfiler runs in parallel in a round-robin style, evenly distributing the list of halos among all processors. Hence, the HaloProfiler will not work in parallel when **halos** is set to single.

## 7.2.2 Profiles

Once the HaloProfiler object has been instantiated, fields can be added for profiling with the `add_profile()` method:

```
hp.add_profile('CellVolume', weight_field=None, accumulation=True)
hp.add_profile('TotalMassMsun', weight_field=None, accumulation=True)
hp.add_profile('Density', weight_field=None, accumulation=False)
hp.add_profile('Temperature', weight_field='CellMassMsun', accumulation=False)
hp.make_profiles()
```

The `make_profiles()` method will begin the profiling.



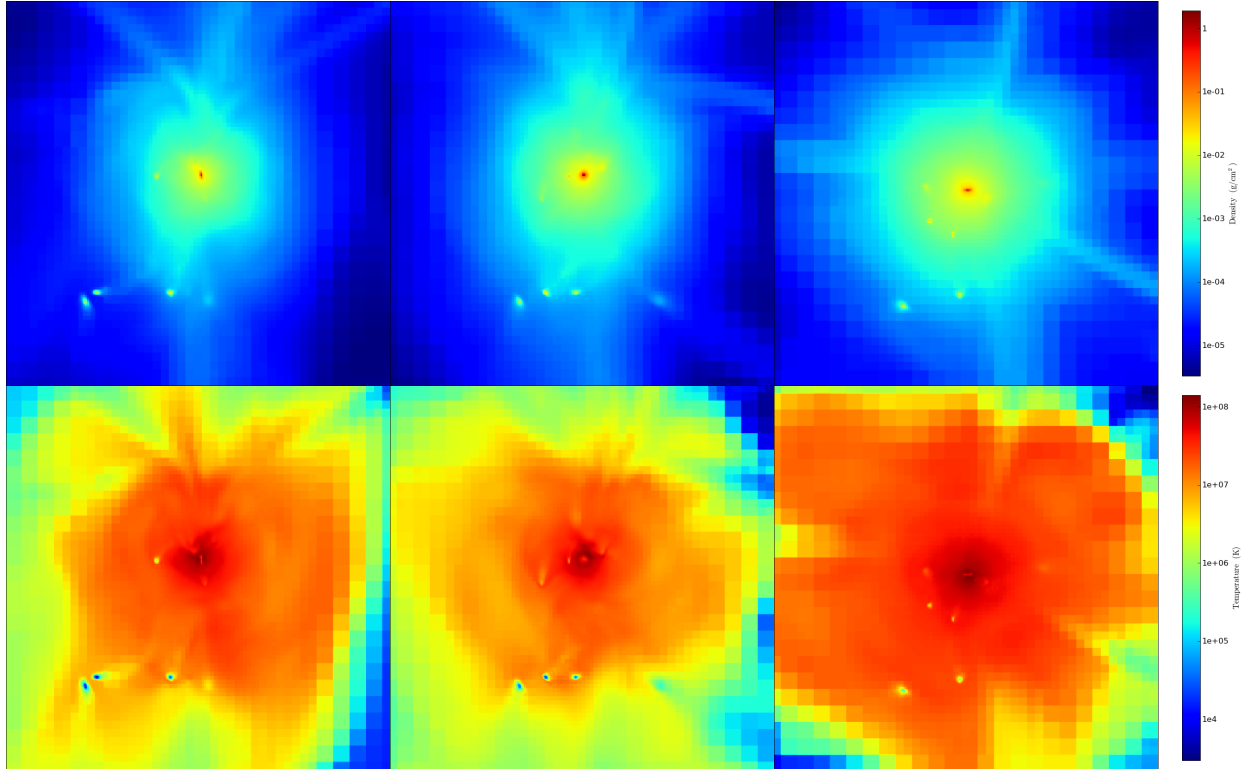
Radial profiles of Overdensity (left) and Temperature (right) for five halos.

### 7.2.3 Projections

The process of making projections is similar to that of profiles:

```
hp.add_projection('Density', weight_field=None)
hp.add_projection('Temperature', weight_field='Density')
hp.add_projection('Metallicity', weight_field='Density')
hp.make_projections(axes=[0, 1, 2], save_cube=True, save_images=True, halo_list="filtered")
```

If `save_cube` is set to `True`, the projection data will be written to a set of hdf5 files in the directory given by **projection\_output\_dir**. The keyword, **halo\_list**, can be used to select between the full list of halos (“all”), the filtered list (“filtered”), or an entirely new list given in the form of a file name. See [Filter Functions](#) for a discussion of filtering halos.



Projections of Density (top) and Temperature, weighted by Density (bottom), in the x (left), y (middle), and z (right) directions for a single halo with a width of 8 Mpc.

## 7.2.4 Halo Filters

Filters can be added to create a refined list of halos based on their profiles or to avoid profiling halos altogether based on information given in the halo list file.

### Filter Functions

It is often the case that one is looking to identify halos with a specific set of properties. This can be accomplished through the creation of filter functions. A filter function can take as many args and kwargs as you like, as long as the first argument is a profile object, or at least a dictionary which contains the profile arrays for each field. Filter functions must return a list of two things. The first is a True or False indicating whether the halo passed the filter. The second is a dictionary containing quantities calculated for that halo that will be written to a file if the halo passes the filter. A sample filter function based on virial quantities can be found in `yt/extensions/HaloFilters.py`.

Halo filtering takes place during the call to `make_profiles()`. The `add_halo_filter()` method is used to add a filter to be used during the profiling:

```
hp.add_halo_filter(HP.VirialFilter, must_be_virialized=True,
                  overdensity_field='ActualOverdensity',
                  virial_overdensity=200,
                  virial_filters=[['TotalMassMsun', '>=', '1e14']],
                  virial_quantities=['TotalMassMsun', 'RadiusMpc'])
```

The addition above will calculate and return virial quantities, mass and radius, for an overdensity of 200. In order to pass the filter, at least one point in the profile must be above the specified overdensity and the virial mass must be at least  $1e14$  solar masses. If the `VirialFilter` function has been added to the filter list, the `HaloProfiler` will make sure

that the fields necessary for calculating virial quantities are added. As many filters as desired can be added. If filters have been added, the next call to `make_profiles()` will filter by all of the added filter functions:

```
hp.make_profiles(filename="FilteredQuantities.out")
```

If the **filename** keyword is set, a file will be written with all of the filtered halos and the quantities returned by the filter functions.

**Note:** If the profiles have already been run, the HaloProfiler will read in the previously created output files instead of re-running the profiles. The HaloProfiler will check to make sure the output file contains all of the requested halo fields. If not, the profile will be made again from scratch.

## Pre-filters

A single dataset can contain thousands or tens of thousands of halos. Significant time can be saved by not profiling halos that are certain to not pass any filter functions in place. Simple filters based on quantities provided in the initial halo list can be used to filter out unwanted halos using the **prefilters** keyword:

```
hp.make_profiles(filename="FilteredQuantities.out",
                 prefilters=["halo['mass'] > 1e13"])
```

Arguments provided with the **prefilters** keyword should be given as a list of strings. Each string in the list will be evaluated with an *eval*.

**Note:** If a VirialFilter function has been added with a filter based on mass (as in the example above), a prefilter will be automatically added to filter out halos with masses greater or less than (depending on the conditional of the filter) a factor of ten of the specified virial mass.

## 7.3 Analyzing an Entire Simulation

*Section author: Britton Smith <britton.smith@colorado.edu>*

The EnzoSimulation class provides a simple framework for performing the same analysis on multiple datasets in a single simulation. At its most basic, an EnzoSimulation object gives you access to a time-ordered list of datasets over the time or redshift interval of your choosing. It also includes more sophisticated machinery for stitching together cosmological datasets to create a continuous volume spanning a given redshift interval. This is the engine that powers the light cone generator (see *light-gone-generator*).

### 7.3.1 EnzoSimulation Options

The only argument required to instantiate an EnzoSimulation is the path to the parameter file used to run the simulation:

```
import yt.extensions.EnzoSimulation as ES
es = ES.EnzoSimulation("my_simulation.par")
```

The EnzoSimulation object will then read through the simulation parameter file to figure out what datasets are available and where they are located. Comment characters are respected, so commented-out lines will be ignored. If no time and/or redshift interval is specified using the keyword arguments listed below, the EnzoSimulation object will create a time-ordered list of all datasets.

**Note:** For cosmological simulations, the interval of interest can be specified with a combination of time and redshift keywords.

The additional keyword options are:

- **initial\_time** (*float*): the initial time in code units for the dataset list. Default: None.
- **final\_time** (*float*): the final time in code units for the dataset list. Default: None.
- **initial\_redshift** (*float*): the initial (highest) redshift for the dataset list. Only for cosmological simulations. Default: None.
- **final\_redshift** (*float*): the final (lowest) redshift for the dataset list. Only for cosmological simulations. Default: None.
- **links** (*bool*): if True, each entry in the dataset list will contain entries, *previous* and *next*, that point to the previous and next entries on the dataset list. Default: False.
- **enzo\_parameters** (*dict*): a dictionary specify additional parameters to be retrieved from the parameter file. The format should be the name of the parameter as the key and the variable type as the value. For example, {'CosmologyComovingBoxSize':float}. All parameter values will be stored in the dictionary attribute, *enzoParameters*. Default: None.
- **get\_time\_outputs** (*bool*): if False, the time datasets, specified in Enzo with the *dtDataDump*, will not be added to the dataset list. Default: True.
- **get\_redshift\_outputs** (*bool*): if False, the redshift datasets will not be added to the dataset list. Default: True.

**Warning:** The EnzoSimulation object will use the *GlobalDir* Enzo parameter to determine the absolute path to the data, so make sure this is set correctly if the data has been moved. If this parameter is not present in the parameter file, the code will look for the data in the current directory.

## 7.3.2 The Dataset List

The primary attribute of an EnzoSimulation object is the dataset list, *allOutputs*. Each list item is a dictionary, containing the time, redshift (if cosmological), and filename of the dataset.

```
>>> es.allOutputs[0]
{'filename': '/Users/britton/EnzoRuns/cool_core_unreasonable/RD0000/RD0000',
 'time': 0.81631644849936602, 'redshift': 99.0}
```

Now, analyzing each dataset is easy:

```
for output in es.allOutputs:
    # load up a dataset
    pf = load(output['filename'])
    # do something!
```

## 7.3.3 Cosmology Splices

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. The *\_create\_cosmology\_splice* method will return such a list:

```
cosmo = es._create_cosmology_splice(
    minimal=True, deltaz_min=0.0, initial_redshift=1.0, final_redshift=0.0)
```

The returned list is of the same format as the *allOutputs* attribute. The keyword arguments are:

- **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.

- **deltaz\_min** (*float*): specifies the minimum  $\Delta z$  between consecutive datasets in the returned list.
- **initial\_redshift** (*float*): the initial (highest) redshift in the cosmology splice list. If none given, the highest redshift dataset present will be used.
- **final\_redshift** (*float*): the final (lowest) redshift in the cosmology splice list. If none given, the lowest redshift dataset present will be used.

The most well known application of this function is the *light cone generator*.

### 7.3.4 Running the HaloProfiler on all Datasets

The following recipe will run the HaloProfiler (see *HaloProfiler*) on all the datasets in one simulation between  $z = 10$  and 0. (cookbook\_simulation\_halo\_profiler.py)

```
1  import yt.extensions.EnzoSimulation as ES
2  import yt.extensions.HaloProfiler as HP
3
4  es = ES.EnzoSimulation("simulation_parameter_file", initial_redshift=10, final_redshift=0)
5
6  # Loop over all dataset in the requested time interval.
7  for output in es.allOutputs:
8
9      # Instantiate HaloProfiler for this dataset.
10     hp = HP.HaloProfiler(output['filename'])
11
12     # Add a virialization filter.
13     hp.add_halo_filter(HP.VirialFilter, must_be_virialized=True,
14                       overdensity_field='ActualOverdensity',
15                       virial_overdensity=200,
16                       virial_filters=[['TotalMassMsun', '>=', '1e14']],
17                       virial_quantities=['TotalMassMsun', 'RadiusMpc'])
18
19     # Add profile fields.
20     hp.add_profile('CellVolume', weight_field=None, accumulation=True)
21     hp.add_profile('TotalMassMsun', weight_field=None, accumulation=True)
22     hp.add_profile('Density', weight_field=None, accumulation=False)
23     hp.add_profile('Temperature', weight_field='CellMassMsun', accumulation=False)
24     # Make profiles and output filtered halo list to FilteredQuantities.out.
25     hp.make_profiles(filename="FilteredQuantities.out")
26
27     # Add projection fields.
28     hp.add_projection('Density', weight_field=None)
29     hp.add_projection('Temperature', weight_field='Density')
30     hp.add_projection('Metallicity', weight_field='Density')
31     # Make projections for all three axes using the filtered halo list and
32     # save data to hdf5 files.
33     hp.make_projections(save_cube=True, save_images=True,
34                       halo_list='filtered', axes=[0,1,2])
35
36 del hp
```



# CONTRIBUTING CODE

`yt` is designed to be accessible to contributions, of both enhancements to the core packages and the library of recipes and scripts for performing common – and not-so-common – tasks.

## 8.1 Bug Fixes

If you have simple bug fixes, please feel free to attach them to a ticket on the [bug tracker](#) (you might have to [register](#) first) or to email them to one of the developers directly. We're always happy to hear about the things we've done wrong, and how you've fixed them!

## 8.2 Licensing

All contributed code must be GPL-compatible; we ask that you consider licensing under the GPL version 3, but we will consider submissions of code that are BSD-like licensed as well. If you'd rather not license in this manner, but still want to contribute, just drop me a line and I'll put a link on the main wiki page to wherever you like!

## 8.3 Fields and Extensions

`yt` comes with a bunch of derived fields. However, if you have constructed some that add interesting analysis quantities, please feel free to send them to one of the developers!

Additionally, if you have a sub-module that extends `yt` in a fun or exciting way, we'd be very happy to include it. Recently we've added light cone generators, halo profilers, and work is even ongoing on a parallel halo finder!

## 8.4 Analysis Code and Examples

Because `yt` can be a bit difficult to become fully acquainted with, we encourage you to share your analysis scripts. Specifically, we will provide you with free repository space to store any analysis scripts that went into the writing of a paper. Through this, we hope to build up a library not only of usage-cases, but of real-world examples of plot generation and data analysis.

If you are interested in submitting your scripts, please contact Matt Turk at [matthewturk@gmail.com](mailto:matthewturk@gmail.com).



# ASKING FOR HELP

If you run into problems with `yt`, you should feel **encouraged** to ask for help – whether this comes in the form of reporting a bug or emailing the mailing list. If something doesn't work for you, it's in everyone's best interests to make sure that it gets fixed.

## 9.1 The Mailing List

The mailing list should be your first stop, every time and always. If you are having a problem, it might be something other people have struggled with and fixed on their own, or it might be a bug – in which case it has to be brought to the attention of the developers!

There are two mailing lists, `yt-users` and `yt-dev`. The first should be used for asking for help, suggesting features and so on, and the latter has more chatter about the way the code is developed and discussions of changes and feature improvements.

If you email `yt-users` asking for help, there are several things you must provide, or else we won't be able to do much:

1. What it is that went wrong, and how you knew it went wrong.
2. A traceback if appropriate – but see *Error Reporting with the Pastebin* for some help with that.
3. If possible, the smallest number of steps that can reproduce the problem.
4. Which version of the code you are using.

When you email the list, providing this information can help the developers understand what you did, how it went wrong, and any potential fixes or similar problems they have seen in the past. Without this context, it can be very difficult to help out!

## 9.2 Installation Issues

If you are having installation issues, and you've read the *Installation* section of the manual, you should *definitely* email the `yt-users` email list. You should provide information about the host, the version of the code you are using, and the output of `yt_install.log` from your installation. We are very interested in making sure that `yt` installs everywhere!

## 9.3 Vanilla Usage Issues

If you're running `yt` without having made any modifications to the code base, please provide as much of your script as you are able to. Submitting both the script and the traceback to the pastebin (as described in *The Pastebin*) is usually sufficient to reproduce the error.

## 9.4 Customization and Scripting Issues

If you have customized `yt` in some way, or created your own plugins file (as described in *The Plugin File*) then it may be necessary to supply both your patches to the source and the plugin file, if you are utilizing something defined in that file.

## 9.5 How To Report A Bug

The first step, when reporting a bug, is to identify the smallest piece of code that reproduces the bug.

To submit a bug report, register an account on the [yt Trac site](#) and submit a [new ticket](#). Alternatively, email the `yt-users` mailing list and we will construct a new ticket in your stead.

## FAQ

### 10.1 Why Python?

Well, the easiest answer is that I knew it, and it met the requirements. The more interesting answer, though, is a combination of things. Python right now has a lot of momentum behind it, particularly in the scientific community. It's easy to compile, portable across many architectures, relatively simple to write C-based extensions for, and it's well-suited to rapid application development. With access to an interpreter, new avenues of data-exploration are opened, and this can lead to much more rapid and interesting analysis.

### 10.2 Where can I learn more about Python?

There are several good, free books about Python available on the web. The best place to start is with the [official tutorial](#), but there's also [Dive Into Python](#), an entire [collection of videos](#) on [ShowMeDo.com](#), and a much more specific guide to using [NumPy](#), which is the backend on which all the math done in yt is based.

As far as books go, the only book I've found to be absolutely indispensable is the [Beazley Book](#).

### 10.3 Who works on yt?

Matthew Turk is the lead developer, but Britton Smith, Jeff Oishi, Dave Collins and Stephen Skory have all made substantive contributions.

### 10.4 What's up with the names?

In the book [Snow Crash](#), yt is Uncle Enzo's messenger. Lagos is the keeper of the data, Raven is a master slicer, and so on. In version 2.0, many of these names will be eliminated when the code base is reorganized.

### 10.5 Are there any restrictions on my use of yt?

yt has been released under Version 3 of the [GNU General Public License](#).

If you found it useful, and have extended it in some meaningful way, of course I'd love to see your contributions so they can be shared with the community. Additionally, if you use yt in a paper, I'd love it if you'd drop me a line to let me know.

## 10.6 How do I know what the units returned are?

This is a very important question. The derived fields – and the native data types – are returned as CGS units, to the best knowledge of the code; but if you see something that looks way off, you should investigate. To see, specifically, what yt is returning for a given field, you can do:

```
print lagos.fieldInfo[some_field].units
```

and it will show you the units that have been assigned to it.

If you are defining your own derived field, you should assume that the units given to the function you define are already in CGS.

That being said, if for some reason yt is unable to determine the correct units for your simulation, it will notify you. It knows how to parse output from all of the versions of Enzo I have used or encountered, and the newest public release is a target platform. However, the Enzo and Orion codebases are so diverse and – at times – fragmented that it is difficult if not impossible to know that all the corner cases have been identified and handled.

## 10.7 What are all these .yt files?

By default, yt attempts to serialize a couple pieces of data that help speed it up in future invocations. Specifically, the entire contents of the hierarchy, the parent-child relationships between the grids, and any projections of the entire volume that are made. Furthermore, objects that have been saved to the hierarchy are stored here as well.

## 10.8 How can I help?

If you find a bug, report it. If you do something cool, write it up. If you find a place to improve the code, send in a patch. We're very interested in contributions! There is a set of [hacking guidelines](#) on the wiki.

## 10.9 Something has gone wrong. What do I do?

Well, first off, double check that you're giving the code what it needs, and not asking it for something it can't provide. Use the `help()` command on an object or a method to get more information.

If you can't figure out what's up, please go ahead and copy the resultant traceback information (the error message it prints out) along with any log files, and either send an email to the `yt-users` mailing list (subscribe first!) or attach them to a ticket at <http://yt.enzotools.org/>. If you are running from within a script, re-run the script with `--paste` on the command line; this will upload the error message to the [pastebin](#) and print a URL. Include the URL in your email message to the mailing list.

## 10.10 How do I specify an axis?

For now, axes are specified by integers – 0,1,2 for x,y,z. In version 2.0 this will probably change to allow for string-identification as well.

## 10.11 Where can I go for support?

I've set up a `yt-users` mailing list. There's more information about it at the [yt homepage](#) and in the section *Asking for Help*.





# YT METHODS

**Warning:** This is a subsection of the chapter in Matthew Turk’s thesis on the development and capabilities of `yt`. Some of the specifics may be out of date, but the mathematical and algorithmic descriptions are still valid.

## 11.1 Introduction

The construction and development of analysis tools acts as a rite of passage for computational astrophysics students. As scientists, the goal is always the same: understanding and examining output of a simulation and then processing it to produce some insight about natural phenomena. However, non-intuitive output formats and the relatively time-consuming process of constructing and testing modules to process these outputs delays the process of generating useful analysis methods and tools. Furthermore, this leads to non-standard analysis tools, which may conceal both bugs and creeping errors. Clearly, a flexible and freely-distributed means of analyzing and exploring data would serve to enhance the scientific process, easing the transition from generating data to *understanding* data.

Adaptive mesh refinement, in particular, usually consists of relatively intuitive but not straightforward data formats. The Enzo code, described in this work, relies on regular, cartesian grid patches consisting of computational elements, and has been used to study a wide variety of astrophysical problems. A competing code called Orion, also built on adaptive mesh refinement technology, is also used to study astrophysical problems, albeit utilizing different underlying solvers and physical models. Both start from the presupposition that at all locations in a given computational domain, all quantities governing physical processes are defined by a set of fields. These data fields act in conjunction to completely describe the state of the simulation.

Several sets of tools exist to handle adaptive mesh refinement data; in the astrophysical community alone, there are several to choose from. Jacques, written in IDL by Tom Abel (<http://jacques.enzotools.org>), acts as a visualization system for Enzo data. VisIt, developed at Lawrence Livermore National Lab, is a 3D visualization suite that can examine both Enzo and Orion data. Volume renderers, such as that presented in [vg06-kaehler], allow for interactive and immersive rendering of adaptive mesh refinement. Additionally, numerous home-grown scripts and modules, developed in isolation, have been designed over the lifetimes of these codes to handle analysis and visualization. However, what was missing was a flexible, lightweight solution built exclusively on freely available open source components. This would allow not only for largely unfettered redistribution, but also a complete examination of every component of the analysis process, from data to plot.

I present here an analysis toolkit I have created called `yt`, which is built exclusively on free and open source components and is unencumbered by heavyweight libraries and licensing servers. I have designed it to be highly modular, with a clear data analysis module distinct from the visualization and plotting modules, and a graphical user interface that builds on rather than supplanting the underlying application programming interface.

`yt` is written primarily in Python, with some computationally expensive routines written in C for speed. Python is an open source, freely-available object-oriented language designed for rapid development and ease of use. Python use is increasingly widespread, both inside and outside the scientific domain, for purposes as diverse as serving dynamic

content online to symbolic math processing. Here we use it to provide transparently parallel analysis and visualization of adaptive mesh refinement simulations of astrophysical phenomena, a task to which it is ideally suited. Additionally, this allows the creation and usage of new analysis modules by users, which can be built on the foundations of the `yt` framework.

The definition of Free Software requires a number of freedoms: the freedom to use, the freedom to inspect, the freedom to give away, and the freedom to modify. These principles serve science well, and all of them serve to improve repeatability and non-locality of results. Not only are all of the components of `yt` Free Software, but the libraries it is built upon are Free Software. Enzo, as well, is Free Software and runs on exclusively Free Software operating systems. In this way, the development of `yt` helps to ensure the entire pipeline of analysis – from the simulation to the paper – is open and available to all parties.

## 11.2 Analysis Requirements

Astrophysical systems are inherently multi-scale, and the formation of primordial stars is a good example. Beginning with cosmological-scale perturbations in the background density of the universe, one must follow the evolution of gas parcels down to the mass scale of the moon to have any hope of resolving the inner structure and thus constrain the mass scale of these stars. The Enzo code, used in this work to simulate the formation of the first stars in the universe, is also used for simulating large-scale galaxy clusters [2007ApJ-671-27H], and galaxy formation and evolution [2009ApJ-696-96W]. These diverse applications require flexible analysis methods that work for broad but shallow refinement regions – as in a turbulence simulation – as well as narrow and deep refinement, as in a primordial star formation simulation.

Approaching from the standpoint of examining slices and projected regions in extremely deep adaptive mesh refinement datasets, `yt` was created to approach the problem of off-screen rendering and scriptable interfaces. To accommodate the relatively diverse computing environments on which Enzo is run, exclusively interactive visualization had to be replaced with a detached method more suited to remote visualization, oftentimes through a job execution queue on a computing cluster. By detaching the user interface from the analysis backend, the architecture was restructured to be a loosely federated system of components. Currently, `yt` is primarily a scripting interface for analysis and visualization, with limited data management capabilities. However, a full graphical user interface for interactive exploration, built on wxPython, remains a crucial part of the toolkit as a whole. These components all interact as modules, and thus can operate completely independently of each other.

Utilizing commodity Python-based packages, `yt` is a fully-featured, adaptable and versatile means of analyzing large-scale astrophysical data. It is based primarily on the library NumPy and it is mostly written in Python. It uses Matplotlib for visualization, and optionally PyTables and wxPython for various sub-tasks. Additionally, several core routines have been written in C for fast numerical computation, and a simple TVTK-based 3D visualization component has been implemented. A community of users and developers has grown around the project; it has been used in several published papers and is now distributed with the Enzo code itself.

The ultimate purpose of `yt` is to provide a high-level interface to data, which will have the side effect of enabling different entry points to `yt` itself. This interface includes the creation of publication-quality plots, as well as a concealment of difficult, multi-step operations. This allows the creation of multiple frontends, as well as recipe-based approaches to analysis script creation. To provide maximum flexibility, as well as a conceptual separation of the different components and tasks to which components can be directed, `yt` is packaged into several sub-packages, for data handling, data analysis, and plotting.

## 11.3 Community Engagement

From its beginning, `yt` has been exclusively free and open source software, and it will never require components that are not open source and freely available. This eliminates dependencies on licensing servers, as well as contributing back to the community any developed technology. The development has been driven, and will continue to be driven, by the pragmatic analysis needs of working scientists.

Furthermore, no implemented features will be hidden from the community at large. This philosophy has served the toolkit well already; the analysis toolkit has been examined by outsiders and minor bugs have been found and corrected. While this provision does not extend to components provided by others, it has served well for the development team, as all new features and components are developed in the open with peer review.

The development of `yt` takes place in a publicly accessible subversion repository with a Trac frontend. Cross-referenced and indexed documentation is available, and automatically updated as changes are made. The source code is entirely commented and extensive programming interface documentation is automatically generated. In order to ease the process of installation, a script is included to install the entire set of dependencies along with the toolkit; furthermore, installations of the toolkit are maintained at several different supercomputing centers, and a binary version for Mac OS X is provided.

This high-level of community involvement and, more importantly, *outreach* enables a broader set of diverse needs and desires to guide the long-term development. Enabling direct technology transfer between users, rather than requiring re-implementation, allows the the community to disentangle the coding process from the scientific process; simultaneously, by making all code public, inspectable and freely available, it can be openly improved and verified. The availability and relatively approachable nature of `yt`, in addition to the inclusion of many simple analysis tasks, reduces the barrier to entry for young scientists; furthermore, by orienting the analysis framework development as a community project, the learning curve for transforming simulation data into publications is greatly reduced.

## 11.4 Data Analysis Layer

The analysis layer, `lagos`, provides several features beyond data access, including extensive analytical capabilities. At its simplest level, `lagos` is used to access the parameters and data in a given data snapshot output from an AMR simulation. Objects are described by physical shapes and orientations, rather than the data structures dictated by the code. This enables an intuitive and physically meaningful entry point to data analysis, rather than a pragmatic approach based on the underlying simulation code base.

### 11.4.1 Physical Objects and Data Selection

One of the difficulties in dealing with rectilinear adaptive mesh refinement data is the fundamental disconnect between the geometries of the grid structure and the objects described by the simulation. One does not expect galaxies to form and be shaped as rectangular prisms; as such, access to physically-meaningful structures must be provided. Therefore, `yt` provides

- Spheres
- Rectangular prisms
- Cylinders (disks)
- Arbitrary regions based on logical operations
- Topologically-connected sets of cells
- Axis-orthogonal and arbitrary-angle rays
- Axis-orthogonal and arbitrary-angle slices
- Arbitrary fixed-resolution grids
- Projected planes

Each of these regional descriptors is presented to the user as a single object, and when accessed the data is returned at the finest resolution available; all overlapping coarse grid cells are removed transparently. This was first implemented as physical structures resembling spheres were to be analyzed, followed by disk-like structures, each of which needed

to be characterized and studied as a whole. By making available these intuitive and geometrically meaningful data selections, the underlying physical structures that they trace become more accessible to analysis and study.

By overloading the normal Python dictionary-like accessor methods, the objects mediate access to data fields defined at every cell. The simple command

```
>>> some_object["Density"]
```

initiates a procedure that begins by examining the current contents of the datastore of object `some_object`, proceeds to read the `Density` field from the disk from those grids from which it is culled, concatenates the individual fields into a single array, and then returns that to the user.

The abstraction layer is such that there are several means of interacting with these three-dimensional objects, each of which is conceptually unified, and which respects a given set of data protocols. Due to the flexibility of Python, as well as the versatility of NumPy, this functionality has been easily exposed in the form of multiple returned arrays of data, which are fast and easily manipulated. Below can be seen the calculation of the angular momentum vector of a sphere, and then the usage of that vector to construct a disk with a height relative to the radius.

```
sp = amr_hierarchy.sphere(center, radius)
print sp["Density"].min()
L_vec = sp.quantities["AngularMomentumVector"]()
my_disk = amr_hierarchy.disk(center, L_vec,
                             radius, radius/100.0)
print my_disk["Density"].min()
```

These objects handle cell-based data fields natively, but are also able to appropriately select and return particles contained within them. This has facilitated the inclusion of an off-the-shelf halo finder (discussed below) which allows users to quantify the clustering of particles within a region.

## 11.4.2 Object Storage

The construction of objects, as well as derived data fields, can often be a computationally expensive task; in particular, clumps found by the contouring algorithm (see *Contour Finding*) and the gravitational binding checks that are used to describe them require a relatively time-consuming set of steps. To save time and enable repeatable analysis, the storage of objects between sessions is essential. Python itself comes with an object serialization protocol called `pickle` that can handle most objects. However, by default the pickle protocol is greedy – it seeks to take all affiliated data. For a given `yt` object, this may include the entire hierarchy, the parameter file, all arrays associated with that object, and even user-space variables. Under the assumption that the data used to generate the fields within a given object will be available the next time the object is accessed, we can reduce the size and scope of the pickling process by designing a means of storing and retrieving these objects across sessions.

Implementing the `__reduce__` method on an object allows the description of a pickling protocol. For all `yt` objects, this protocol has been specified as a description in physical space of the object itself; this usually constitutes replicating the arguments to the constructor – the radius and center of a sphere, for instance. For extracted objects based on indices of parent objects, the indices are stored as well. Once the protocol has been executed, binary data designed to reconstruct the object is stored – either in a single, standalone file or in the parameter file-affiliated data store, ending in the extension `yt`.

The biggest obstacle to retrieving an object is the affiliation of an object with a given parameter file. At their most base level, a parameter file can be described by a path. However, while this works for a single instantiation of a `yt` session, often between sessions data will be moved – between supercomputing centers, or across mounted external hard drives, or even within a given computing center to a different storage system. A means of addressing, or at least uniquely identifying, parameter files is necessary to ensure uniform access across instances of an analysis session. An absolute path, while unique, is not necessarily invariant. To this end, the `basename` (the final element in the absolute path), the simulation time, and the creation time of the simulation output (`CurrentTimeIdentifier` in Enzo) are used to identify a given static output. An MD5 hash is generated of these three items, which is then used as a key for

the parameter file. By this means, collisions between different parameter files (rather than copies of a single parameter file) are made extremely unlikely.

Upon retrieval of the object, the key is handed to a parameter file storage object. This object keeps track of all instantiated parameter files; whenever a new parameter file object is instantiated, its hash is generated and compared against the set of existing parameter files. If a match is found, the current path is compared to the path being used during instantiation, and the path in the data store is updated as necessary. If the parameter file is new to the system, it is inserted. By this means, the locations of all known parameter files are kept as up to date as possible; this is by no means a foolproof system, but it works in most cases.

### 11.4.3 Grid Patches

The Enzo and Orion codes are based around “patch-based” refinement. For every set of cells flagged to be refined, a minimally-enclosing box is selected for refinement. This grid patch is then used as a container and as a computational element, and cell data output to disk is grouped into the parent grid patches. In addition to field and particle data, each possesses a set of attributes that describe its position, its relationship to other grids, and its cell spacing:

- Parent(s)
- Unique identifier
- Level number
- Left edge
- Right edge
- Dimensions
- Children

The cell spacing is easily computed as  $dx_{i,j,k} = (LE_{i,j,k} - RE_{i,j,k})/D_{i,j,k}$  where  $i, j, k$  are the axes,  $LE$  is the left edge,  $RE$  is the right edge and  $D$  is the number of cells along that axis. The regions covered by grid patches are not uniquely covered; higher-level child grids overlap with cells in their parent grid, and often that data needs to be removed to ensure that only the highest resolution data is used for analysis purposes. For this purpose, `yt` provides an affiliated `child_mask` for every grid; this is a boolean array with identical dimensionality, but wherever a child grid covers a cell, that cell’s index in the mask is set to zero. Everywhere the grid contains the finest data available, the mask cells are set to one. This caching of the locations of child cells enables rapid selection of cells where the data is already the most refined available.

### 11.4.4 Data Fields

The model for handling data, and processing fundamental data fields into new fields describing derived quantities, is designed to be built on top of an object model. Presupposing the existence of object `sphere`, we can access the field `Density` by accessing it in a dictionary like fashion. On top of this, we can build automatically recursive field generators that depend on other fields. All fields, including derived fields, are allowed to be defined by either a component of a data file, or a function that transforms one or more other fields, thus allowing multiple layers of definition to exist, and allowing the user to extend the existing field set as needed.

By defining simple functions that automatically operate via array operations, generating derived fields is straightforward and fast. For instance, a field such as the magnitude of the velocity in a cell

$$V = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

can be defined independently of the source of the data:

```
def VelocityMagnitude(field, data):  
    return (data["x-velocity"]**2.0 +  
            data["y-velocity"]**2.0 +  
            data["z-velocity"]**2.0)**0.5
```

Each operation acts independently on each element of the source data fields; this preserves the abstraction of fields as undifferentiated sets of cells, when in fact those cells could be distributed spatially over the entire dataset, with varying cell widths and varying grid levels.

Once a function is defined, it is added to a global field container that contains not only the fields, but a set of metadata about each field – the unit specifier, the unit specifier for projected versions of that field, and any implicit or explicit requirements for that field. Field definitions can require that certain parameters be provided (such as a height vector, a center point, a bulk velocity and so on) or, most powerfully, that the data object has some given characteristic. This is typically applied to ensure that data is given in a spatial context; for finite difference solutions, such as calculating the gradient or divergence of a set of fields, `yt` allows the derived field to mandate that the input data provided in a three-dimensional structure. Furthermore, when specifying that some data object be provided in three dimensions, a number of buffer cells can be specified as well; the returned data structure will then have those buffer cells taken from neighboring grids. This enables higher-order methods to be used in the generation of fields, for instance when a given finite difference stencil extends beyond the computational domain of a single grid patch.

## 11.4.5 Two-Dimensional Data Representations

In order to make images and plots, `yt` has several different classes of two-dimensional data representations, all of which can be turned into images. Each of these objects generates a list of variable-resolution points, which are then passed into a C-based pixelization routine that transforms them into a fixed-resolution buffer, defined by a width, a height, and physical boundaries of the source data.

### Slices

The simplest means of examining data is through the usage of grid-axis aligned slices through the dataset. This has several benefits - it is easy to calculate which grids and which cells are required to be read off disk (and most data formats allow for easy striding of data off disk, which reduces this operation's IO overhead) and the process of stepping through a given dataset is relatively easy to automate.

To construct a set of data points representing a slice, all grids intersected by the slice are first examined, and then the index of the cell desired is generated

$$\text{floor}(p - v_i)/dx$$

where  $p$  is the position of the slice,  $v_i$  is the coordinate of the left-edge of the grid along the axis of the slice and  $dx$  is the cell spacing of the grid along the axis of the slice. By this process we construct a set of data points defined as  $(x_p, dx_p, y_p, dy_p, v)$  where  $p$  indicates that this is in the image plane rather than in the global coordinates of the simulation, and  $v$  is the value of the field selected; furthermore, every returned  $(x_p, dx_p, y_p, dy_p, v)$  point does not overlap with any points where  $dx < dx_p$  or  $dy < dy_p$ ; thus each point is the finest resolution available.

To construct an image buffer, these cells are pixelized and placed into a fixed-resolution array, defined by  $(x_{p,\min}, x_{p,\max}, y_{p,\min}, y_{p,\max})$ . Every pixel in the image plane is iterated over, and any cells that overlap with it are deposited into every pixel  $I_{ij}$  as:

$$\alpha = A_c/A_p$$
$$\alpha v \rightarrow I_{ij}$$

where  $\alpha$  is an attempt to anti-alias the output image plane, to account for misalignment in the image and world coordinate systems and  $A_c$  and  $A_p$  are the areas of the cell and pixel respectively. Anti-aliasing can be disabled, as well.

## Projections

The nature of adaptive mesh refinement is such that one often wishes to examine either the sum of values along a given sight-line or a weighted-average along a given sight-line. `yt` provides an algorithm for generating line integrals in an adaptive fashion, such that every returned  $(x_p, dx_p, y_p, dy_p, v)$  point does not contain data from any points where  $dx < dx_p$  or  $dy < dy_p$ ; the alternative being a binned histogram, where fixed-width cells are defined perpendicular to the line of sight and then data is filled into those cells. By providing this list of finest-resolution data points in a projected domain, images of any width can be constructed essentially instantaneously; conversely, however, the projection process takes longer, for reasons described below.

To obtain the finest points available, the grids are iterated over in order of the level of refinement – first the coarsest and then proceeding to the finest levels of refinement. The process of projecting a grid is slightly variant, dependent on the desired output from the projection. For weighted averages,

$$\begin{aligned} V_{ij} &= \sum_n v_{ijn} w_{ijn} dl \\ W_{ij} &= \sum_n w_{ijn} dl \end{aligned}$$

where  $V_{ij}$  is the output value at every cell in the image plane,  $v_{ijn}$  is every cell in the grid’s data field,  $w_{ijn}$  is the weight field at every cell in the grid’s data field, and  $dl$  is the path length through a single cell. Note that because this process is conducted on a grid-by-grid basis, and the  $dl$  does not change within a given grid, this term can be moved outside of the sum. In the limit of an unweighted integration,  $W_{ij}$  is set to 1.0, rather than to the evaluation of the sum. Furthermore, a mask of child cells is reduced with a logical and operation along the axis of projection; any cell where this mask is “False” has data of a higher refinement level available to it. This grid is then compared against all grids on the same level of refinement with which it overlaps; the flattened  $x$  and  $y$  position arrays are compared via integer indexing and any collisions are combined. This process is repeated with data from coarser grids that has been identified as having subsequent data available to it; each coarse cell is then added to the  $r^2$  cells on the current level of processing, where  $r$  is the refinement factor. At this point, all cells in the array of data for the current level where the reduced child mask is “True” are removed from subsequent processing, as they are part of the final output of the projection. All cells where the child mask is “False” are retained to be processed on the next level. In this manner, we create a cascading refinement process, where only two levels of refinement have to be compared at a given time.

When the entire data hierarchy has been processed, the final flattened arrays of  $V_p$  and  $W_p$  are divided to construct the output data value

$$v(x, y) = V(x, y) / W(x, y)$$

which is kept as the weighted average value along the axis of projection. In the case of direct integration, note that  $W(x, y)$  is in fact unity, so this is a pass-through operation. Once this process is completed, the projection object respects the same data protocol, and can be plotted in the same way, as an ordinary slice.

## Cutting Planes

At some length scales in star formation problems, gas is likely to collapse into a disk, which is often not aligned with the axes of the simulation. By slicing along the axes, patterns such as spiral density waves could be missed, and ultimately go unexamined. In order to better visualize off-axis phenomena, `yt` is able to create images misaligned with the axes.

A cutting plane is an arbitrarily-aligned plane that transforms the intersected points into a new coordinate system such that they can be pixelized and made into a publication-quality plot. Identifying the data that is transformed into the image, at some arbitrary angle to the disk, is a two-step process.

A central point and a single normal vector are required; this normal vector is taken as normal to the desired image plane. This leaves a degree of freedom for rotation of the image plane about the normal vector and through the central point. A minimization procedure is conducted to determine the appropriate “North” vector in the image plane:

$$\begin{aligned} \mathbf{p}_x &= \mathbf{a}_0 \times \mathbf{n} \\ \mathbf{p}_y &= \mathbf{n} \times \mathbf{p}_x \\ \mathbf{d} &= -\mathbf{c} \cdot \mathbf{n} \end{aligned}$$



where  $\mathbf{a}_0$  is the axis with which the normal vector ( $\mathbf{n}$ ) has the greatest cross product,  $\mathbf{c}$  is the vector to the center point of the plane, and  $\mathbf{d}$  is the inclination vector. From this we construct two matrices, the rotation matrix:

$$R = \begin{pmatrix} p_{xi} & p_{xj} & p_{xk} \\ p_{yi} & p_{yj} & p_{yk} \\ n_i & n_j & n_k \end{pmatrix}$$

and its inverse, which are used to rotate coordinates into and out of the image plane, respectively. Grids are identified as being intersected by the cutting plane through fast array operations on their boundaries. We define a new array,  $D$ , where

$$D_{ij} = \mathbf{v}_{ji} \cdot \mathbf{d}$$

where the index  $i$  is over each grid and the index  $j$  refers to which of the eight grid vertices ( $\mathbf{v}$ ) of the grid is being examined. Grids are accepted if all three components of every  $D_j$  is of identical sign:

$$\text{all}(D_j < 0) \text{ or } \text{all}(D_j > 0).$$

Upon identification of the grids that are intersected by the cutting plane, we select data points by examining the distance of the cell-center to the plane, and selecting points where

$$|\mathbf{p} \cdot \mathbf{n} + \mathbf{d}| < \frac{\sqrt{dx^2 + dy^2 + dz^2}}{2}.$$

This generates a small number of false positives (from regarding a cell as a sphere rather than a rectangular prism), which are removed during the pixelization step when creating a plot. Each data point is then rotated into the image plane via the rotation matrix:

$$\begin{aligned} \mathbf{p} \cdot \mathbf{p}_x &\rightarrow x_p \\ \mathbf{p} \cdot \mathbf{p}_y &\rightarrow y_p. \end{aligned}$$

This technique requires a new pixelization routine, in order to ensure that the correct cells are taken and placed on the plot, which requires an additional set of checks to determine if the cell intersected with the image plane. The process here is similar to the standard pixelization procedure, described above, with the addition of the rotation step. Defining  $d = \sqrt{dx^2 + dy^2 + dz^2}$ , every data point where  $(x_p \pm d, y_p \pm d)$  is within the bounds of the image is examined by the pixelization routine for overlap of the data point with a pixel in the output buffer. Every potentially intersecting pixel is then iterated over and the coordinates  $(x_i, y_i, 0)$  of the image buffer are rotated via the inverse rotation matrix back to the world coordinates  $(x', y', z')$ . These are then compared against the  $(x, y, z)$  of this original datapoint. If all three conditions

$$\begin{aligned} |x - x'| &< dx \\ |y - y'| &< dy \\ |z - z'| &< dz \end{aligned}$$

are satisfied, the data value from the cell is deposited in that image buffer pixel. An unfortunate side effect of the relatively complicated pixelization procedure, as well as the strict intersection-based inclusion, is that the process of antialiasing is non-trivial and computationally expensive. As such, these images often appear quite jagged at cell-pixel boundaries. Additionally, utilizing the same transformation and pixelization process, overlaying velocity vectors is trivially accomplished and such a process is included in the toolkit.

## 11.4.6 Contour Finding

Visual inspection of simulations provides a simple method of identifying distinct hydrodynamic regions; however, a quantitative approach must be taken to describe those regions. Specifically, distinct collapsing regions can be identified by locating topologically-connected sets of cells. The nature of adaptive mesh refinement, wherein a given set of cells may be connected across grid and refinement boundaries, requires traversing grid and resolution boundaries.



Unfortunately, while locating connected sets inside a single-resolution grid is a straightforward but non-trivial problem in recursive programming, extending this in an efficient way to hierarchical datasets can be problematic. To that end, the algorithm implemented in `yt` checks on a grid-by-grid basis, utilizing a buffer zone of cells at the grid boundary to communicate set identification. The algorithm for identifying these sets is a recursive and iterative process:

1. Identify grids to be considered, such as from `AMRSphereBase` object
2. Give unique identification numbers to all finest-level cells within the desired contour ( $v_{\min} \leq v \leq v_{\max}$ )
3. Construct expandable queue of grids to be examined
  - (a) Give unique identification number to all coarse-cells in considered grid within desired contour ( $v_{\min} \leq v \leq v_{\max}$ )
  - (b) Obtain buffer zone of one cell-width, including contour IDs
  - (c) Recursively examine all cells identified as contour members
    - i. Update contour ID to be the maximum of 26 neighboring cells
    - ii. If current contour ID is greater than original contour ID, repeat until it is not
    - iii. Notify all neighboring cells with contour ID less than current contour ID to re-examine neighbors and update
  - (d) Flush contour IDs in buffer zone to originating grids
  - (e) If any buffer zones contour IDs have changed during this process, re-order queue such that the next grids to be examined are originating grids of changed contour IDs
4. Reorder contour IDs such that the largest contours have the lowest numbers
5. Return extracted contour objects

Any contour that crosses into the buffer zones mandates a reconsideration of all grids that intersect with the currently considered grid. This process is expensive, as it operates recursively, but ensures that all contours are automatically joined.

Once contours are identified, they are split into individual derived objects that are returned to the user. This presents an integrated interface for generating and analyzing topologically-connected sets of related cells. This method was used in [2009ApJ-691-441S] to study fragmentation of collapsing gas clouds, specifically to examine the gravitational boundedness of these clouds and the length and density scales at which fragmentation occurs.

To determine whether or not an object is bound, we evaluate the inequality

$$\sum_{i=1}^N \frac{m_i v_i^2}{2} < \sum_{i=1}^{N-1} \sum_{j=i+1}^N \frac{G m_i m_j}{r}$$

where  $n$  is the number of cells in the identified contour. The left hand side of this equation is the total kinetic energy in the object; if desired, the internal thermal energy ( $nkT / (\gamma - 1)$ ) can also be added to this term. This code has been written to run either in a hand-coded C module or on the graphics processor, using NVIDIA's CUDA framework (<http://www.nvidia.com/cuda/>) via the PyCUDA (<http://mathematician.de/software/pycuda>) package. Moving the calculation onto the graphics card speeds the calculation up nearly ideally by two orders of magnitude. This allows for binding checks on extremely large datasets in a manageable amount of time.

### 11.4.7 Fixed Resolution Grids

The particular structures of multi-resolution data can impede certain classes of algorithms. To address this need, the creation of fixed-resolution (and three-dimensional) arrays of data must be easy and accessible. However, unless the entire region under consideration is contained within a single grid patch, it can be difficult to construct these arrays. The method included in `yt` for creating these “covering grids” is to select all grids within a given rectangular prism.

These grids are then iterated over, starting on the coarsest level, and used to fill in each point in the new array. Only cells that intersect with the array are considered, and any grid cell that intersects with any cell within the covering grid is included, as long as the child mask for that cell indicates no finer data is available. By this method, the entire covering grid is filled in with the finest cells available to it. This can be utilized for generating ghost zones, as well as for minimum covering grids out of many single-resolution grids that are disjoint in the domain.

However, because coarse cells are duplicated across all cells in the (possibly finer-resolution) covering grid with which they intersect, this can lead to unwanted resolution artifacts. To combat this, a “smoothed” covering grid object is also available. This object is filled in completely at all levels  $l < L$  where  $L$  is the level at which the covering grid is being extracted. Once a given level has been filled in, the grid is trilinearly interpolated to the next level, and then all new data points from grids at that level replace existing data points. This method is suitable for generating smoothed multi-resolution grids and constructing vertex-centered data, as used in Section *Immersive Visualization with VTK*.

### 11.4.8 Multi-dimensional Profiles

Distributions of data within the space of other variables are often necessary when examining and analyzing data. For instance, in a collapsing gas cloud, examining the average temperature with increasing radius from a central location provides a convenient means of examining the process of collapse, as well as the effective equation of state. To conduct this sort of analysis, typically a multi-dimensional histogram is constructed, wherein the values in every bin are weighted averages of some additional quantity. In `yt`, the term “profile” is used to describe any weighted average or distribution of a variable with respect to a second, independent variable. Such uses include a histogram of temperature with respect to density, a radial profile of molecular hydrogen fraction, and a radius, temperature, and velocity phase diagram. With the usage of the open-source, 3D rendering engine S2PLOT (<http://astronomy.swin.edu.au/s2plot/index.php?title=S2PLOT>), these profiles can have up to three independent variables.

One can imagine profiles serving two different purposes: to show the average value of a variable at a fixed location in the phase space of a set of independent variables, or for the distribution of a variable with respect to a set of independent variables. The first step is that of binning or histogramming. We define up to three axes of comparison, which will be designated  $x$ ,  $y$ , and  $z$ , but should not be confused with the spatial axes. These are discretized into  $x_0 \dots x_n$  where  $n$  is the number of bins along the specified axis. Indices  $j$  for each value among the set of points being profiled are then generated along each axis such that

$$x_j \leq v_i < x_{j+1}.$$

These indices are then used to calculate the weighted average in each bin:

$$V_j = \frac{\sum_{i=1}^N v_i w_i}{\sum_{i=1}^N w_i}$$

where  $V_j$  is now the average value in bin  $j$  in our weighted average, and the  $N$  points are selected such that their index along the considered axis is  $j$ . If we wish to examine multiple dimensions, we simply mandate that in all dimensions, the index of all the points used in the average is the index of the bin into which values are being placed. To conduct a non-averaged distribution, the weights are all set to 1.0 in the numerator, and the sum in the denominator is not calculated. This allows, for example, the examination of mass distribution in a plane defined by chemo-thermal quantities.

### 11.4.9 Parallel Analysis

As the capabilities of supercomputers grow, the size of datasets grows as well. Most standalone codes are not parallelized; the process is time-consuming, complicated, and error-prone. Therefore, the disconnect between simulation time and data analysis time has grown ever larger. In order to meet these changing needs, `yt` has been modified to run

in parallel on multiple independent processing units on a single dataset. Specifically, utilizing the Message Passing Interface (MPI) via the MPI4Py (<http://code.google.com/p/mmpi4py/>) module, a lightweight, NumPy-native wrapper that enables natural access to the C-based routines for interprocess communication, the code has been able to subdivide datasets into multiple decomposed regions that can then be analyzed independently and joined to provide a final result. A primary goal of this process has been to preserve at all times the API, such that the user can submit an unchanged serial script to a batch processing queue, and the toolkit will recognize it is being run in parallel and distribute tasks appropriately.

The tasks in `yt` that require parallel analysis can be divided into two broad categories: those tasks that act on data in an unordered, uncorrelated fashion (such as weighted histograms, summations, and some bulk property calculation), and those tasks that act on a decomposed domain (such as halo finding and projection).

## Unordered Analysis

To parallelize unordered analysis tasks, a set of convenience functions have been implemented utilizing an initialize/finalize formalism; this abstracts the entirety of the analysis task as a transaction. Signaling the beginning and end of the analysis transaction sets in motion several procedures, defined by the analysis task itself, that handle the initialization of data objects and variables and that combine information across processors. These are abstracted by the base class `ParallelAnalysisInterface`, which implements several different methods useful for parallel analysis. By this means, the intrusion of parallel methods and algorithms into previously serial tasks is kept to a minimum; invasive changes are typically not necessary.

This transaction follows several steps:

1. Call `get_grids` to obtain list of grids to process
2. Iterator calls `object._initialize_parallel`
3. Object processes each grid
4. Iterator calls `object._finalize_parallel` and raises `StopIteration`.

Inside the routine `get_grids` the iterator decomposes the full collection of grids into chunks based on the organization of the datasets on disk. Implementation of the parallel analysis interface mandates that objects implement two gatekeeper functions, `object._initialize_parallel` and `object._finalize_parallel`. These two functions are allowed to broadcast and communicate with other processors. At the end of the finalization step, the object is expected to be identical on all processors. This enables scripts to be run identically in parallel and in serial. For unordered analysis, this process results in close-to-ideal scaling with the number of processors.

Upon initialization, `ParallelAnalysisInterface` determines which sets of data will be processed by which processors. In order to decompose a task across processors, a means of assigning grids to processors is required. For spatially oriented-tasks (such as projections) this is simple and accomplished through the decomposition of some spatial domain. For unordered analysis tasks, the clear means by which grids can be selected is through a minimization of file input overhead. The process of reading a single set of grid data from disk can be outlined as:

1. Open file
2. Seek to grid data position
3. Read data
4. Close file

However, in the case of “packed” Enzo data, as well as all Orion data, multiple grids are written to a single file. If we know the order in which these grids are written, we can consolidate several data reads into a single operation:

1. Open file
2. For each grid
  - (a) Seek to grid position

- (b) Read each field
- 3. Close file

If we know the means by which the grids and fields are ordered on disk, we can simplify the seeking requirements and instead read in large sweeps across the disk. By further pre-allocating all necessary memory, this becomes a single operation that can be accomplished in one “sweep” across each file. By allocating as many grids from a single “grid output” file on a single processor, this procedure can be used to minimize file overhead on each processor.

## Spatial Decomposition

MPI provides a means of decomposing an arbitrary region across a given number of processors. Because of the inherently spatial nature of the adaptive projection algorithm implemented in `yt`, parallelization requires decomposition with respect to the image plane; however, future revisions of the algorithm may allow for unordered grid projection. To project in parallel, the computational domain is divided such that the image plane is distributed equally among the processors; each component of the image plane is then used to construct rectangular prisms along the entire line of sight. Each processor is thus allocated a rectangular prism of dimensions

$$(L_i, L_j, L_d)$$

where the axes have been rotated such that the line of sight of the projection is the third dimension,  $L_i L_j$  is constant across processors, and  $L_d$  is the entire computational domain along the axis of projection. Following the projection algorithm, each processor will then have a final image plane set of points, as per usual:

$$(x_p, dx_p, y_p, dy_p, v)$$

but subject to the constraints that all points are contained within the rectangular prism as prescribed by the image plane decomposition. At the end of the projection step all processors join their image arrays, which are guaranteed to contain only unique points.

Enzo and Orion utilize different file formats, but both are designed to output a single file per processor with all constituent grids computed on that processor localized to that file. Unfortunately, both codes conduct “load balancing” operations on the computational domain, so processors are not necessarily guaranteed to have spatially localized grids; this results in the output format not being spatially decomposed, but rather unordered. As a result, this method of projection does not scale as well as desired, because each processor is likely to have to read grid datasets from many files. Despite that, the communication overhead is essentially irrelevant, because the processors only need to communicate the end of the projection process, to share their non-overlapping final result with all other processors in the computational group.

### 11.4.10 Halo Finding

In cosmological hydrodynamic simulations, dark matter particles and gas parcels are coupled through gravitational interaction. Furthermore, dark matter dominates gravitational interaction on all but the smallest scales. Dark matter particles act as a collisionless fluid, and are the first component of the simulation to collapse into identifiable structures; as such, they can be used effectively to identify regions of structure formation.

The HOP algorithm [eishut98] is an effective and tested means of identifying collapsed dark matter halos in a simulation, and has been a part of the Enzo code distribution for some time. Typically an Enzo simulation is allowed to execute to completion, an entire dataset is loaded into memory, and then the HOP algorithm processes the entire domain. This process is memory-intensive, and requires that the entire dataset be loaded into a single computer. It is not inherently parallel and thus does no domain decomposition. The output from this is a single list of halos and the associated densities, masses, particle identifiers, positions, and so on. The HOP algorithm works by assigning a density to every particle; each particle then “hops” to its most dense neighbor. Each set of particles sharing a most dense neighbor is then called a group, and any groups with a density below the minimum density threshold (a free parameter) is removed from the final list of groups. These groups are then rejoined along boundaries.

Including this code inside yt, as a means of abstracting away compilation and data access, was trivial; however, to do so the input to HOP was generalized to be an arbitrary three-dimensional data source. As a result, the HOP algorithm can now be applied on subsets of the domain. By decomposing the domain into multiple tiles with a buffer region, the HOP algorithm can be run on multiple processors, with a final “join” operation performed to construct a full halo list. Any halo whose most dense point is located within the buffer zone is cut, as those halos should be found on neighboring tiles.

However, the free parameter in this calculation is that of the size of the buffer zone. A balance must be struck between identification of objects and memory requirements; clearly, based on the means of identifying, if a halo happens to reside within the buffer zone of a tile and it is greater in spatial extent than that of the buffer zone, it will be truncated on both sides. This problem is mitigated by the particular set of problems where a parallel halo finder is needed. These problems, with more particles than can fit in the main memory of a standard HPC cluster node, are likely to be extremely large physical domain problems, with relatively small halos. In the circumstances where a large simulation has very large halos, greater than the size of the buffer zone, this method would be unsuitable, as it would split the identification of halos over the buffer zones. This situation could arise, for instance, in a relatively small physical domain simulation with extremely high resolution dark matter particles, where micro-halos could be missed by this technique.

The goal of having a parallel halo finder is to reduce the memory and processing time overhead for large simulations; by distributing the identification of dark matter halos across multiple, independent processors, we gain an increased efficiency, but we must construct creative means of communication. As such, the halo data container objects themselves have been transformed into “proxy” objects, transparently communicating requests for information.

## 11.5 Plotting and Visualization Layer

The plotting layer, `yt.raven`, can plot one-, two- and three-dimensional histograms of quantities, allowing for weighting and binning of those results. A set of pixelization routines have been written in C to provide a means of taking a set of variable-size pixels and constructing a uniform grid of values, suitable for fast plotting in Matplotlib. Applicable cases include non-axially perpendicular planes, allowing for oblique slices to be plotted and displayed with publication-quality rendering. Callbacks are available for overlaying analytic solutions, grid-patch boundaries, vectors, contours, and arbitrary annotation.

## 11.6 Constraints of Scale

In order to manage simulations consisting of hundreds of thousands of discrete grid patches – as well as their attendant grid cell values – bottlenecks have been located and eliminated using the `cProfile` module. Additionally, the practice of storing data about simulation outputs between instantiation of the Python objects has been extended; this speeds subsequent startups, and enables faster response times. Because very large hierarchies consume substantial time during the parsing and instantiation of attributes, a core set of data about the geometry and structure of the grid objects is stored in a fast array format, eliminating the need to repeatedly convert text values to internal floating point representation.

Enzo data is written in one of three ways, the most efficient way being via the Hierarchical Data Format (HDF5) with a single file per processor that the simulation was run on. To limit the effect that disk access has on the process of loading data, hand-written wrappers to the HDF5 have been inserted into the code. These wrappers are lightweight, and operate on a single file at a time, loading data in the order it has been written to the disk. The package PyTables was used for some time, but the instantiation of the object hierarchy was found to be too much overhead for the brief and well-directed access desired.

## 11.7 Frontends and Interfaces

`yt` was originally intended to be used from the command line, and images to be viewed either in a web browser or via an X11 connection that forwarded the output of an image viewer. However, a happy side-effect of this architecture, as well as the versatile Matplotlib “Canvas” interface, is that the `yt` API, designed to have a single interface to analysis tasks, is easily accessed and utilized by different interfaces. By ensuring that this API is stable and flexible, GUIs, web-interfaces, and command-line scripts can be constructed to perform common tasks.

Not all environments have access to the same level of interactivity. For large-scale datasets, being able to interact with the data through a scripting interface enables submission to a batch processing queue, which enables appropriate allocation of resources. For smaller datasets, the process of interactively exploring datasets via graphical user interfaces, exposing analytical techniques not available to an offline interface, is extremely worthwhile, as it can be highly immersive.

The canonical graphical user interface is written in wxPython, and presents to the user a hierarchical listing of data objects: static outputs from the simulation, as well as spatially-oriented objects derived from those outputs. The tabbed display pane shows visual representations of these objects in the form of embedded Matplotlib figures, as seen in Figure [ref{fig:yt:reason\\_bds}](#).

An interface to the interactive Matplotlib `pylab` interface, via IPython, has been prepared. This enables the user to generate plots that are thematically linked, and thus display a uniform spatial extent. Further enhancements to this IPython interface, via the profile system, have been targeted for the next release.

## 11.8 Embedding `yt` Inside Enzo

An outstanding problem in the analysis of large scale data is that of the disk; while data can be written to the disk, read back, and then analyzed in an arbitrary fashion, this process is not only slow but requires substantial intermediate disk space for a substantial quantity of data that will undergo severely reductionist analysis. To address this problem, the typical solution is to insert analysis code, generation of derived quantities, images, and so forth, into the simulation code. However, the usual means of doing this is through either a substantial hand-written framework that attempts to account for every analysis task, or a limited framework that only handles very limited analysis tasks.

Furthermore, by enabling in-line analysis, the relative quantity of analysis output is substantially greater than that enabled by disk-mediated analysis. Removing numerous large files dumped to disk as a prerequisite for conducting analysis and generating visualization allows for a much more favorable ratio of data to analyzed data. For a typical Population III star formation simulation, the size of the data dumps can be as much as 10 gigabytes per timestep; however, the relative amount of information that can be gleaned from these outputs is significantly smaller. Using smaller data output mechanisms as well as more clever streaming methods can improve this ratio; however, by enabling in-line analysis, images of the evolution of a collapsing Population III halo can be output at every single update of the hydrodynamical time, allowing for true “movies” of star formation to be produced. By allowing for the creation and exporting of radial profiles and other analytical methods, this technique opens up vast avenues for analysis while simulations are being conducted, rather than afterward.

The Python/C API allows for passage of data in-memory to an instance of the Python interpreter; by embedding a Python interpreter within each running Enzo MPI task, Enzo is able to pass existing data to a newly spawned `yt` analysis task, and thus disintermediate the disk completely. While this currently works for many relatively simple tasks, it is not currently able to decompose data spatially; as we are constrained by the parallel nature of the Enzo domain decomposition, we attempt to avoid passing data between MPI tasks. This means if a grid is owned by MPI task 1, it will not be passed to MPI task 2 during the analysis stage.



## 11.9 Generalization to Other AMR Codes

As mentioned above, `yt` was designed to handle and analyze data output from the AMR code Enzo. The entire codebase has been ported to work equally well with data from other AMR codes, beginning with the Orion code in use at the University of California, Berkeley. However, different codes make separate sets of assumptions about outputted data, and this must be generalized to be non-Enzo specific. In this process, a balance had to be struck between generalizing data reading and specifications, as well as simplicity and speed. This led to a minimally invasive set of changes, which have been put into place.

The primary architectural change that had to be made was generalizing the means by which data fields were recognized and handled by `yt`. Orion, specifically, stores a different set of state vectors than Enzo. For instance, momentum replaces velocity. To accommodate this, while retaining identical sets of derived fields, a new hierarchy of derived field containers was created: the base set of fields that are “universal,” the Enzo-specific fields, and the Orion-specific fields. The code-specific field containers are responsible for accepting raw data output by the simulation and converting that into a format that the “universal” field set can understand. Unit conversion, as well as transformation of state vectors, and additionally dealing with different assumptions about cell-face and cell-centered field information. Implementing these field containers following the “Borg” design pattern, wherein all instances of a class share a single state, enabled all derived fields, regardless of how generated, to be shared across all data output types and instances.

In the future, `yt` will be expanded to handle and analyze other adaptive mesh refinement codes. Work has begun to port it to handle data output by the FLASH code; a major difficulty in doing so, however, is the handling of the FLASH data format. Unlike both the Enzo and Orion codes, FLASH uses an octree, cell-based refinement scheme. Two ways forward are obvious: either each refined cell is assigned its own grid patch, or a volume segmentation algorithm can be executed to place rectangular prisms in refined regions, thus identifying grid patches.

By providing a unified interface to multiple, often competing, AMR codes, we will be able to utilize similar, if not identical, analysis scripts and algorithms, which will enable direct comparison of results between groups and across methods. Analyzing multiple datasets of identical phenomena at a single time with a single analysis framework is an important and powerful means of comparison across methods and scientific collaborations. Furthermore, utilizing identical means of data access allows for conversion of data between groups for subsequent analysis and re-simulation. Through this method, the results and methods of computation can be verified and compared.

### 11.10 Immersive Visualization with VTK

Visualizing multi-resolution three-dimensional datasets requires careful and detailed methods. While `yt` makes no claims to be a complete solution for such visualization, it provides hooks for exporting data as well as utilizing external libraries for three-dimensional visualization.

A VTK-based frontend has been implemented, utilizing the Traits technology and the TVTK library from Enthought, Inc (<http://www.enthought.com/>). Traits is a rapid application development environment that provides for semi-static typing of variables. This provides the ability to rapidly generate GUIs, as well as validation of input and notification based on change of state of variables.

The TVTK library provides for the construction of multi-resolution structured grid objects called `vtkHierarchicalBoxDataSets`, which are processed as a group rather than as discrete, unique elements. To enable this computation, I created a patch to expose the functionality of the `vtkHierarchicalBoxDataSet` to a scripting interface; this was then exposed to the user and interactive widgets provided for manipulation and creation of contour sets (using the marching cubes algorithm) and planes that cut the volume at arbitrary angles.

We are confined to a maximum of twelve levels due to the precision of the VTK positioning mechanism; attempting to position with finer than single-precision coordinates results in overlapping and indistinguishable elements. In order to expose the deepest hierarchies (with many levels of refinement) a subsection must be excised and presented to the library. This consists of an extraction of all grids confined by a box, defined by  $(x_0, y_0, z_0) \dots (x_1, y_1, z_1)$  and  $K_n \dots K_{n+12}$ , where the coordinates define the left and right edges and the  $K$  variable refers to the level of the base

grid presented to VTK. We scale the left and right edges of the grids in this subregion

$$\begin{aligned} (L - L_{\min}) &\rightarrow L_s \\ r^{l-l_0} (R - L_{\min}) &\rightarrow R_s \end{aligned}$$

where  $L$  and  $R$  are the sets of  $(x_0, y_0, z_0)$  and  $(x_1, y_1, z_1)$ ,  $r$  is the refinement factor,  $l$  is the level,  $l_0$  is the first level of the extraction and  $L_s$  and  $R_s$  are the final scaled values. The grids from the coarsest level are replaced with a smoothed minimal covering box, which may incorporate data from lower levels. This enables us to have a base “medium” into which the higher-resolution levels are placed, rather than multiple disjoint root-level grids. However, by providing this coordinate conversion in both directions, locations in the base data set can be referenced in a straightforward manner.

VTK does not provide the same quality of visualization for AMR data that other solutions do; however, it provides a valuable and flexible means of exploring data, and one that is free and open source. As such, it is currently the preferred direction for future ventures into immersive visualization with `yt`. Furthermore, because it is a base library with a structured approach to visualizing data, it can be used as a basis for more complicated rendering schemes. Unfortunately, because those schemes likely require a more complicated data structure, the overlap may be minimal.

The VTK camera system is straightforward and easy to manipulate; the interface between `yt` and VTK has been equipped with a means of recording, manipulating, playing back and saving camera paths based on points of motion. The user navigates from position to position by whatever means they desire, takes a “snapshot” of the current camera position and orientation, and then specifies how many points on the line they desire. By interpolating the rotation and translation between fixed camera positions, a smooth path of arbitrary frame frequency can be generated and exported to other systems of visualization. Currently only linear interpolation between points is supported; higher-order interpolation would produce smoother camera paths.

## 11.11 Community Involvement

I have conducted the vast majority of development on `yt`, accounting for 816 of the 968 version control “commits” of the 52,000 lines of code comprising `yt` (and several included but external packages) as of the end of February, 2009. However, in recent months, as distribution of the toolkit has increased and as the user base has increased, a substantial uptick in user involvement and submitted development has occurred. In particular, several of the developments discussed here have been explored and implemented by users, including the streamlined halo analyzer, the light cone generation, the parallel halo finder, and the original implementation of the clump finding process, based on the contour finding primitives.

The public face to `yt` is that of a web page (<http://yt.enzotools.org/>), with integrated source control system, ticket and bug tracker, wiki pages, mailing list, recipe book, and “pastebin” of code snippets. By specifying a command line option to any script utilizing `yt` libraries, users can upload error messages and scripts to a central location, where they can be examined, commented on, improved, and discussed.

Currently `yt` is being developed at four different institutions across the United States, and has users in at least ten different institutions worldwide. The first official release (`yt-1.0`) was bundled with Enzo 1.5, and the next release is being prepared by a six-person team of developers writing documentation, fixing bugs, adding features, and providing support for other users. The availability of Python, the simplified all-in-one installation script and the growing user community are clearly factors in this growth of usage; hopefully, in the future, the project will become less centralized and more of a community effort.

## 11.12 Future Directions

As the capabilities of `yt` expand, the ability to extend it to perform new tasks expands as well. By publishing `yt`, and generalizing it to work on multiple AMR codebases, I hope it will foster collaboration and community efforts toward understanding astrophysical problems and physical processes, while enabling reproducible research. The roadmap for



yt has several key milestones; the first of which will be a substantially rewritten set of documentation and the announcement of the general usability of the parallel analysis tasks. Further tasks include better, higher-level interfaces; an expanded scripting interface to yt, and in addition larger-scale “recipes” which would provide easier entry points to analysis and visualization.

One of the weakest aspects of yt is that of time-series analysis. Currently, individual parameter files must be examined and instantiated; this process has been eased by a variety of “recipes” for instantiation and analysis over a set, but unfortunately it is still hobbled by an awkward interface and the tethering of data objects to individual parameter files. By disconnecting data objects from the hierarchy, time-series analysis would become much more tractable; this would enable the construction of time series outputs, composed of multiple static outputs or a single set of “streaming” outputs. These time series objects could be affiliated with data objects assigned a fixed set of parameters defining their selection region, but a varying time component.

By extending the ability to generate synthetic observations, yt will become of greater use for the verification of astrophysical simulations. The ultimate product should be that of telescope-simulated images; ideally, these images could be subjected to identical scrutiny and analysis as those taken directly from telescopes. The prospects for utilizing the same framework for generation of simulated images as well as arbitrary analysis are exciting.



# API DOCUMENTATION

This is a compilation of documentation about the internal data objects of `yt`. It has been separated into sections based on purpose and its location within the code base. It's not meant as a replacement for narrative documentation, but instead as a supplement.

Contents:

## 12.1 `yt.lagos` Native AMR Data Structures

These are data structures for interacting with the various AMR platforms that `yt` understands and can analyze.

### 12.1.1 `yt.lagos.OutputTypes` Output Types

**class** `EnzoStaticOutput` (*filename*, *data\_style=None*, *parameter\_override=None*, *conversion\_override=None*)  
Enzo-specific output, set at a fixed time.

This class is a stripped down class that simply reads and parses *filename* without looking at the hierarchy. *data\_style* gets passed to the hierarchy to pre-determine the style of data-output. However, it is not strictly necessary. Optionally you may specify a *parameter\_override* dictionary that will override anything in the parameter file and a *conversion\_override* dictionary that consists of {fieldname : conversion\_to\_cgs} that will override the #DataCGS.

**cosmology\_get\_units** ()

Return an Enzo-fortran style dictionary of units to feed into custom routines. This is typically only necessary if you are interacting with fortran code.

**get\_parameter** (*parameter*, *type=None*)

Gets a parameter not in the parameterDict.

**has\_key** (*key*)

Returns true or false

**keys** ()

Returns a list of possible keys, from `_units`, `parameters` and `_conversion_factors`

**class** `OrionStaticOutput` (*plotname*, *paramFilename=None*, *fparamFilename=None*, *data\_style=7*, *para-noia=False*)

This class is a stripped down class that simply reads and parses, without looking at the Orion hierarchy.

@todo:

@param filename: The filename of the parameterfile we want to load @type filename: String

need to override for Orion file structure.

the paramfile is usually called “inputs” and there may be a fortran inputs file usually called “probin” plotname here will be a directory name as per BoxLib, `data_style` will be one of

Native IEEE (not implemented in yt) ASCII (not implemented in yt)

**has\_key** (*key*)

Returns true or false

**keys** ()

Returns a list of possible keys, from `_units`, `parameters` and `_conversion_factors`

**class StaticOutput** (*filename, data\_style=None*)

Base class for generating new output types. Principally consists of a *filename* and a *data\_style* which will be passed on to children.

**has\_key** (*key*)

Returns true or false

**keys** ()

Returns a list of possible keys, from `_units`, `parameters` and `_conversion_factors`

## 12.1.2 yt.lagos.HierarchyTypes Grid Hierarchies

**class EnzoHierarchy** (*pf, data\_style=None*)

This is the grid structure as Enzo sees it, with some added bonuses. It’s primarily used as a class factory, to generate data objects and access grids.

It should never be created directly – you should always access it via calls to an affiliated `EnzoStaticOutput`.

On instantiation, it processes the hierarchy and generates the grids.

**export\_boxes\_pv** (*filename*)

Exports the grid structure in partview text format.

**export\_particles\_pb** (*filename, filter=1, indexboundary=0, fields=None, scale=1.0*)

Exports all the star particles, or a subset, to pb-format *filename* for viewing in partview. Filters based on `particle_type=*filter*`, `particle_index>=*indexboundary*`, and exports *fields*, if supplied. Otherwise, index, position(x,y,z). Optionally *scale* by a given factor before outputting.

**findMax** (*\*args, \*\*kwargs*)

Returns (value, center) of location of maximum for a given field.

**find\_max** (*field, finestLevels=True*)

Returns (value, center) of location of maximum for a given field.

**find\_min** (*field*)

Returns (value, center) of location of minimum for a given field

**find\_point** (*coord*)

Returns the (objects, indices) of grids containing an (x,y,z) point

**find\_ray\_grids** (*coord, axis*)

Returns the (objects, indices) of grids that an (x,y) ray intersects along *axis*

**find\_slice\_grids** (*coord, axis*)

Returns the (objects, indices) of grids that a slice intersects along *axis*

**find\_sphere\_grids** (*center, radius*)

Returns objects, indices of grids within a sphere

**get\_box\_grids** (*left\_edge, right\_edge*)

Gets back all the grids between a left edge and right edge

**get\_data** (*node*, *name*)  
Return the dataset with a given *name* located at *node* in the datafile.

**get\_smallest\_dx** ()  
Returns (in code units) the smallest cell size in the simulation.

**load\_object** (*name*)  
Load and return an object from the *data\_file* using the Pickle protocol, under the name *name* on the node /Objects.

**print\_stats** ()  
Prints out (stdout) relevant information about the simulation

**save\_data** (*array*, *node*, *name*, *set\_attr=None*, *force=False*, *passthrough=False*)  
Arbitrary numpy data will be saved to the region in the datafile described by *node* and *name*. If data file does not exist, it throws no error and simply does not save.

**save\_object** (*obj*, *name*)  
Save an object (*obj*) to the *data\_file* using the Pickle protocol, under the name *name* on the node /Objects.

**select\_grids** (*level*)  
Returns an array of grids at *level*.

**class OrionHierarchy** (*pf*, *data\_style=7*)

**export\_boxes\_pv** (*filename*)  
Exports the grid structure in partview text format.

**export\_particles\_pb** (*filename*, *filter=1*, *indexboundary=0*, *fields=None*, *scale=1.0*)  
Exports all the star particles, or a subset, to pb-format *filename* for viewing in partview. Filters based on *particle\_type=\*filter\**, *particle\_index>=\*indexboundary\**, and exports *fields*, if supplied. Otherwise, index, position(x,y,z). Optionally *scale* by a given factor before outputting.

**findMax** (\*args, \*\*kwargs)  
Returns (value, center) of location of maximum for a given field.

**find\_max** (*field*, *finestLevels=True*)  
Returns (value, center) of location of maximum for a given field.

**find\_min** (*field*)  
Returns (value, center) of location of minimum for a given field

**find\_point** (*coord*)  
Returns the (objects, indices) of grids containing an (x,y,z) point

**find\_ray\_grids** (*coord*, *axis*)  
Returns the (objects, indices) of grids that an (x,y) ray intersects along *axis*

**find\_slice\_grids** (*coord*, *axis*)  
Returns the (objects, indices) of grids that a slice intersects along *axis*

**find\_sphere\_grids** (*center*, *radius*)  
Returns objects, indices of grids within a sphere

**get\_box\_grids** (*left\_edge*, *right\_edge*)  
Gets back all the grids between a left edge and right edge

**get\_data** (*node*, *name*)  
Return the dataset with a given *name* located at *node* in the datafile.

**get\_smallest\_dx** ()  
Returns (in code units) the smallest cell size in the simulation.

**load\_object** (*name*)  
Load and return an object from the *data\_file* using the Pickle protocol, under the name *name* on the node /Objects.

**print\_stats** ()  
Prints out (stdout) relevant information about the simulation

**readGlobalHeader** (*filename*, *paranoid\_read*)  
read the global header file for an Orion plotfile output.

**save\_data** (*array*, *node*, *name*, *set\_attr=None*, *force=False*, *passthrough=False*)  
Arbitrary numpy data will be saved to the region in the datafile described by *node* and *name*. If data file does not exist, it throws no error and simply does not save.

**save\_object** (*obj*, *name*)  
Save an object (*obj*) to the *data\_file* using the Pickle protocol, under the name *name* on the node /Objects.

**select\_grids** (*level*)  
Returns an array of grids at *level*.

**class AMRHierarchy** (*pf*)

**export\_boxes\_pv** (*filename*)  
Exports the grid structure in partview text format.

**export\_particles\_pb** (*filename*, *filter=1*, *indexboundary=0*, *fields=None*, *scale=1.0*)  
Exports all the star particles, or a subset, to pb-format *filename* for viewing in partview. Filters based on *particle\_type=\*filter\**, *particle\_index>=\*indexboundary\**, and exports *fields*, if supplied. Otherwise, index, position(x,y,z). Optionally *scale* by a given factor before outputting.

**findMax** (*\*args*, *\*\*kwargs*)  
Returns (value, center) of location of maximum for a given field.

**find\_max** (*field*, *finestLevels=True*)  
Returns (value, center) of location of maximum for a given field.

**find\_min** (*field*)  
Returns (value, center) of location of minimum for a given field

**find\_point** (*coord*)  
Returns the (objects, indices) of grids containing an (x,y,z) point

**find\_ray\_grids** (*coord*, *axis*)  
Returns the (objects, indices) of grids that an (x,y) ray intersects along *axis*

**find\_slice\_grids** (*coord*, *axis*)  
Returns the (objects, indices) of grids that a slice intersects along *axis*

**find\_sphere\_grids** (*center*, *radius*)  
Returns objects, indices of grids within a sphere

**get\_box\_grids** (*left\_edge*, *right\_edge*)  
Gets back all the grids between a left edge and right edge

**get\_data** (*node*, *name*)  
Return the dataset with a given *name* located at *node* in the datafile.

**get\_smallest\_dx** ()  
Returns (in code units) the smallest cell size in the simulation.

**load\_object** (*name*)  
Load and return an object from the *data\_file* using the Pickle protocol, under the name *name* on the node /Objects.

**print\_stats** ()  
Prints out (stdout) relevant information about the simulation

**save\_data** (*array*, *node*, *name*, *set\_attr=None*, *force=False*, *passthrough=False*)  
Arbitrary numpy data will be saved to the region in the datafile described by *node* and *name*. If data file does not exist, it throws no error and simply does not save.

**save\_object** (*obj*, *name*)  
Save an object (*obj*) to the *data\_file* using the Pickle protocol, under the name *name* on the node /Objects.

**select\_grids** (*level*)  
Returns an array of grids at *level*.

### 12.1.3 yt.lagos.BaseGridType Grid Types

**class EnzoGridBase** (*id*, *filename=None*, *hierarchy=None*)  
Class representing a single Enzo Grid instance.

Returns an instance of EnzoGrid with *id*, associated with *filename* and *hierarchy*.

**clear\_all** ()  
Clears all datafields from memory and calls `clear_derived_quantities()`.

**clear\_all\_grid\_references** ()  
This clears out all references this grid has to any others, as well as the hierarchy. It's like extra-cleaning after `clear_data`.

**clear\_data** ()  
Clear out the following things: *child\_mask*, *child\_indices*, all fields, all field parameters.

**clear\_derived\_quantities** ()  
Clears coordinates, *child\_indices*, *child\_mask*.

**convert** (*datatype*)  
This will attempt to convert a given unit to cgs from code units. It either returns the multiplicative factor or throws a `KeyError`.

**find\_max** (*field*)  
Returns value, index of maximum value of *field* in this grid

**find\_min** (*field*)  
Returns value, index of minimum value of *field* in this grid

**get\_data** (*field*)  
Returns a field or set of fields for a key or set of keys

**get\_field\_parameter** (*name*, *default=None*)  
This is typically only used by derived field functions, but it returns parameters used to generate fields.

**get\_global\_startindex** ()  
Return the integer starting index for each dimension at the current level.

**get\_position** (*index*)  
Returns center position of an *index*

**has\_field\_parameter** (*name*)  
Checks if a field parameter is set.

**has\_key** (*key*)  
Checks if a data field already exists.

**save\_object** (*name, filename=None*)  
Save an object. If *filename* is supplied, it will be stored in a **:module:'shelve'** file of that name. Otherwise, it will be stored via `yt.lagos.AMRHierarchy.save_object()`.

**set\_field\_parameter** (*name, val*)  
Here we set up dictionaries that get passed up and down and ultimately to derived fields.

**set\_filename** (*filename*)  
Intelligently set the filename.

**class OrionGridBase** (*LeftEdge, RightEdge, index, level, filename, offset, dimensions, start, stop, paranoia=False*)

**clear\_all** ()  
Clears all datafields from memory and calls `clear_derived_quantities()`.

**clear\_all\_grid\_references** ()  
This clears out all references this grid has to any others, as well as the hierarchy. It's like extra-cleaning after `clear_data`.

**clear\_data** ()  
Clear out the following things: `child_mask`, `child_indices`, all fields, all field parameters.

**clear\_derived\_quantities** ()  
Clears coordinates, `child_indices`, `child_mask`.

**convert** (*datatype*)  
This will attempt to convert a given unit to cgs from code units. It either returns the multiplicative factor or throws a `KeyError`.

**find\_max** (*field*)  
Returns value, index of maximum value of *field* in this grid

**find\_min** (*field*)  
Returns value, index of minimum value of *field* in this grid

**get\_data** (*field*)  
Returns a field or set of fields for a key or set of keys

**get\_field\_parameter** (*name, default=None*)  
This is typically only used by derived field functions, but it returns parameters used to generate fields.

**get\_position** (*index*)  
Returns center position of an *index*

**has\_field\_parameter** (*name*)  
Checks if a field parameter is set.

**has\_key** (*key*)  
Checks if a data field already exists.

**save\_object** (*name, filename=None*)  
Save an object. If *filename* is supplied, it will be stored in a **:module:'shelve'** file of that name. Otherwise, it will be stored via `yt.lagos.AMRHierarchy.save_object()`.

**set\_field\_parameter** (*name, val*)  
Here we set up dictionaries that get passed up and down and ultimately to derived fields.

**class AMRGridPatch** (*id, filename=None, hierarchy=None*)



**clear\_all()**  
Clears all datafields from memory and calls `clear_derived_quantities()`.

**clear\_all\_grid\_references()**  
This clears out all references this grid has to any others, as well as the hierarchy. It's like extra-cleaning after `clear_data`.

**clear\_data()**  
Clear out the following things: `child_mask`, `child_indices`, all fields, all field parameters.

**clear\_derived\_quantities()**  
Clears coordinates, `child_indices`, `child_mask`.

**convert(*datatype*)**  
This will attempt to convert a given unit to cgs from code units. It either returns the multiplicative factor or throws a `KeyError`.

**find\_max(*field*)**  
Returns value, index of maximum value of *field* in this grid

**find\_min(*field*)**  
Returns value, index of minimum value of *field* in this grid

**get\_data(*field*)**  
Returns a field or set of fields for a key or set of keys

**get\_field\_parameter(*name*, *default=None*)**  
This is typically only used by derived field functions, but it returns parameters used to generate fields.

**get\_position(*index*)**  
Returns center position of an *index*

**has\_field\_parameter(*name*)**  
Checks if a field parameter is set.

**has\_key(*key*)**  
Checks if a data field already exists.

**save\_object(*name*, *filename=None*)**  
Save an object. If *filename* is supplied, it will be stored in a **module:'shelve'** file of that name. Otherwise, it will be stored via `yt.lagos.AMRHierarchy.save_object()`.

**set\_field\_parameter(*name*, *val*)**  
Here we set up dictionaries that get passed up and down and ultimately to derived fields.

## 12.2 yt.lagos Physical and Derived Data Objects

### 12.2.1 yt.lagos.BaseDataTypes Data Containers and Physical Objects

yt provides a number of data containers, defined such that they satisfy a logical need. Each of these provides only the finest-resolution cells, unless an option is available to restrict the levels from which they draw, as is the case with `AMRCoveringGrid`.

Each of these implements the same primary protocol - all data values can be accessed dictionary-style:

```
>>> object["Density"]
>>> object["Density"].max()
```

For more information about objects, see *Object Methodology* and *Using and Manipulating Objects and Fields*.

## Base Classes

**class** **AMRData** (*pf, fields, \*\*kwargs*)

Generic AMRData container. By itself, will attempt to generate field, read fields (method defined by derived classes) and deal with passing back and forth field parameters.

Typically this is never called directly, but only due to inheritance. It associates a `StaticOutput` with the class, sets its initial set of fields, and the remainder of the arguments are passed as `field_parameters`.

**clear\_data** ()

Clears out all data from the AMRData instance, freeing memory.

**convert** (*datatype*)

This will attempt to convert a given unit to cgs from code units. It either returns the multiplicative factor or throws a `KeyError`.

**get\_field\_parameter** (*name, default=None*)

This is typically only used by derived field functions, but it returns parameters used to generate fields.

**has\_field\_parameter** (*name*)

Checks if a field parameter is set.

**has\_key** (*key*)

Checks if a data field already exists.

**save\_object** (*name, filename=None*)

Save an object. If *filename* is supplied, it will be stored in a **:module:'shelve'** file of that name. Otherwise, it will be stored via `yt.lagos.AMRHierarchy.save_object()`.

**set\_field\_parameter** (*name, val*)

Here we set up dictionaries that get passed up and down and ultimately to derived fields.

**class** **AMR1DData** (*pf, fields, \*\*kwargs*)

Bases: `yt.lagos.BaseDataTypes.AMRData`, `yt.lagos.BaseDataTypes.GridPropertiesMixin`

**clear\_data** ()

Clears out all data from the AMRData instance, freeing memory.

**convert** (*datatype*)

This will attempt to convert a given unit to cgs from code units. It either returns the multiplicative factor or throws a `KeyError`.

**get\_field\_parameter** (*name, default=None*)

This is typically only used by derived field functions, but it returns parameters used to generate fields.

**has\_field\_parameter** (*name*)

Checks if a field parameter is set.

**has\_key** (*key*)

Checks if a data field already exists.

**save\_object** (*name, filename=None*)

Save an object. If *filename* is supplied, it will be stored in a **:module:'shelve'** file of that name. Otherwise, it will be stored via `yt.lagos.AMRHierarchy.save_object()`.

**select\_grids** (*level*)

Return all grids on a given level.

**set\_field\_parameter** (*name, val*)

Here we set up dictionaries that get passed up and down and ultimately to derived fields.

**class** **AMR2DData** (*axis, fields, pf=None, \*\*kwargs*)  
 Bases: `yt.lagos.BaseDataTypes.AMRData`, `yt.lagos.BaseDataTypes.GridPropertiesMixin`, `yt.lagos.ParallelTools.ParallelAnalysisInterface`

Prepares the AMR2DData, normal to *axis*. If *axis* is 4, we are not aligned with any axis.

**clear\_data** ()  
 Clears out all data from the AMRData instance, freeing memory.

**convert** (*datatype*)  
 This will attempt to convert a given unit to cgs from code units. It either returns the multiplicative factor or throws a `KeyError`.

**get\_data** (*fields=None*)  
 Iterates over the list of fields and generates/reads them all.

**get\_field\_parameter** (*name, default=None*)  
 This is typically only used by derived field functions, but it returns parameters used to generate fields.

**has\_field\_parameter** (*name*)  
 Checks if a field parameter is set.

**has\_key** (*key*)  
 Checks if a data field already exists.

**interpolate\_discretize** (*LE, RE, field, side, log\_spacing=True*)  
 This returns a uniform grid of points between *LE* and *RE*, interpolated using the nearest neighbor method, with *side* points on a side.

**save\_object** (*name, filename=None*)  
 Save an object. If *filename* is supplied, it will be stored in a **module:‘shelve’** file of that name. Otherwise, it will be stored via `yt.lagos.AMRHierarchy.save_object()`.

**select\_grids** (*level*)  
 Return all grids on a given level.

**set\_field\_parameter** (*name, val*)  
 Here we set up dictionaries that get passed up and down and ultimately to derived fields.

**class** **AMR3DData** (*center, fields, pf=None, \*\*kwargs*)  
 Bases: `yt.lagos.BaseDataTypes.AMRData`, `yt.lagos.BaseDataTypes.GridPropertiesMixin`

Returns an instance of AMR3DData, or prepares one. Usually only used as a base class. Note that *center* is supplied, but only used for fields and quantities that require it.

**clear\_data** ()  
 Clears out all data from the AMRData instance, freeing memory.

**convert** (*datatype*)  
 This will attempt to convert a given unit to cgs from code units. It either returns the multiplicative factor or throws a `KeyError`.

**cut\_region** (*field\_cuts*)  
 Return an `InLineExtractedRegion`, where the grid cells are cut on the fly with a set of *field\_cuts*.

**extract\_connected\_sets** (*field, num\_levels, min\_val, max\_val, log\_space=True, cumulative=True, cache=False*)  
 This function will create a set of contour objects, defined by having connected cell structures, which can then be studied and used to ‘paint’ their source grids, thus enabling them to be plotted.

**extract\_region** (*indices*)  
 Return an `ExtractedRegion` where the points contained in it are defined as the points in *this* data object with the given *indices*.

**get\_field\_parameter** (*name*, *default=None*)

This is typically only used by derived field functions, but it returns parameters used to generate fields.

**has\_field\_parameter** (*name*)

Checks if a field parameter is set.

**has\_key** (*key*)

Checks if a data field already exists.

**paint\_grids** (*field*, *value*, *default\_value=None*)

This function paints every cell in our dataset with a given *value*. If *default\_value* is given, the other values for the given in every grid are discarded and replaced with *default\_value*. Otherwise, the field is mandated to ‘know how to exist’ in the grid.

Note that this only paints the cells *in the dataset*, so cells in grids with child cells are left untouched.

**save\_object** (*name*, *filename=None*)

Save an object. If *filename* is supplied, it will be stored in a **module:‘shelve’** file of that name. Otherwise, it will be stored via `yt.lagos.AMRHierarchy.save_object()`.

**select\_grids** (*level*)

Return all grids on a given level.

**set\_field\_parameter** (*name*, *val*)

Here we set up dictionaries that get passed up and down and ultimately to derived fields.

**class FakeGridForParticles** (*grid*)

Mock up a grid to insert particle positions and radii into for purposes of confinement in an AMR3DData.

## 1D Data Containers

Each of these inherits from `AMR1DData`, and so has all the member functions defined there.

**class AMROrthoRayBase** (*axis*, *coords*, *fields=None*, *pf=None*, *\*\*kwargs*)

Bases: `yt.lagos.BaseDataTypes.AMR1DData`

Dimensionality is reduced to one, and an ordered list of points at an (x,y) tuple along *axis* are available.

## 2D Data Containers

**class AMRSliceBase** (*axis*, *coord*, *fields=None*, *center=None*, *pf=None*, *node\_name=False*, *\*\*kwargs*)

Bases: `yt.lagos.BaseDataTypes.AMR2DData`

AMRSlice is an orthogonal slice through the data, taking all the points at the finest resolution available and then indexing them. It is more appropriately thought of as a slice ‘operator’ than an object, however, as its field and coordinate can both change.

Slice along *axis**How do I specify an axis?*, at the coordinate *coord*. Optionally supply fields.

**reslice** (*coord*)

Change the entire dataset, clearing out the current data and slicing at a new location. Not terribly useful except for in-place plot changes.

**shift** (*val*)

Moves the slice coordinate up by either a floating point value, or an integer number of indices of the finest grid.

**class AMRCuttingPlaneBase** (*normal*, *center*, *fields=None*, *node\_name=None*, *\*\*kwargs*)

Bases: `yt.lagos.BaseDataTypes.AMR2DData`

AMRCuttingPlane is an oblique plane through the data, defined by a normal vector and a coordinate. It attempts to guess an ‘up’ vector, which cannot be overridden, and then it pixelizes the appropriate data onto the plane without interpolation.

The Cutting Plane slices at an oblique angle, where we use the *normal* vector and the *center* to define the viewing plane. The ‘up’ direction is guessed at automatically.

```
class AMRProjBase (axis, field, weight_field=None, max_level=None, center=None, pf=None, source=None,
                  node_name=None, field_cuts=None, serialize=True, **kwargs)
    Bases: yt.lagos.BaseDataTypes.AMR2DData
```

AMRProj is a projection of a *field* along an *axis*. The field can have an associated *weight\_field*, in which case the values are multiplied by a weight before being summed, and then divided by the sum of that weight.

### 3D Data Containers

```
class AMRSphereBase (center, radius, fields=None, pf=None, **kwargs)
    Bases: yt.lagos.BaseDataTypes.AMR3DData
```

A sphere of points

The most famous of all the data objects, we define it via a *center* and a *radius*.

```
class AMRRegionBase (center, left_edge, right_edge, fields=None, pf=None, **kwargs)
    Bases: yt.lagos.BaseDataTypes.AMR3DData
```

AMRRegions are rectangular prisms of data.

We create an object with a set of three *left\_edge* coordinates, three *right\_edge* coordinates, and a *center* that need not be the center.

```
class AMRCylinderBase (center, normal, radius, height, fields=None, pf=None, **kwargs)
    Bases: yt.lagos.BaseDataTypes.AMR3DData
```

We can define a cylinder (or disk) to act as a data object.

By providing a *center*, a *normal*, a *radius* and a *height* we can define a cylinder of any proportion. Only cells whose centers are within the cylinder will be selected.

```
class AMRGridCollection (center, grid_list, fields=None, pf=None, **kwargs)
    Bases: yt.lagos.BaseDataTypes.AMR3DData
```

An arbitrary selection of grids, within which we accept all points.

By selecting an arbitrary *grid\_list*, we can act on those grids. Child cells are not returned.

```
class AMRCoveringGridBase (level, left_edge, right_edge, dims, fields=None, pf=None, num_ghost_zones=0,
                          use_pbar=True, **kwargs)
    Bases: yt.lagos.BaseDataTypes.AMR3DData
```

Covering grids represent fixed-resolution data over a given region. In order to achieve this goal – for instance in order to obtain ghost zones – grids up to and including the indicated level are included. No interpolation is done (as that would affect the ‘power’ on small scales) on the input data.

The data object returned will consider grids up to *level* in generating fixed resolution data between *left\_edge* and *right\_edge* that is *dims* (3-values) on a side.

```
flush_data (field=None)
```

Any modifications made to the data in this object are pushed back to the originating grids, except the cells where those grids are both below the current level *and* have child cells.

```
class AMRSmoothedCoveringGridBase (*args, **kwargs)
    Bases: yt.lagos.BaseDataTypes.AMRCoveringGridBase
```

```
class ExtractedRegionBase (base_region, indices, force_refresh=True, **kwargs)
```

Bases: `yt.lagos.BaseDataTypes.AMR3DDData`

ExtractedRegions are arbitrarily defined containers of data, useful for things like selection along a baryon field.

## 12.2.2 `yt.lagos.DerivedQuantities` Derived Quantities

All of these are accessed via the `.quantities[]` object, and feeding it the function name without the leading underscore. For instance:

```
my_sphere.quantities["TotalMass"]()
```

They all accept the `lazy_reader` option, which governs whether the calculation is performed out of core or not. For more information, see [Derived Quantities](#).

**`__AngularMomentumVector`** (*data*)

This function returns the mass-weighted average angular momentum vector.

**`__BaryonSpinParameter`** (*data*)

This function returns the spin parameter for the baryons, but it uses the particles in calculating enclosed mass.

**`__BulkVelocity`** (*data*)

This function returns the mass-weighted average velocity in the object.

**`__CenterOfMass`** (*data*)

This function takes no arguments and returns the location of the center of mass of the *non-particle* data in the object.

**`__Extrema`** (*data*, *fields*)

This function returns the extrema of a set of fields

**Parameter** *fields* – A field name, or a list of field names

**`__IsBound`** (*data*, *truncate=True*, *include\_thermal\_energy=False*)

This returns whether or not the object is gravitationally bound

**Parameters**

- *truncate* – Should the calculation stop once the ratio of gravitational:kinetic is 1.0?
- *include\_thermal\_energy* – Should we add the energy from ThermalEnergy on to the kinetic energy to calculate binding energy?

**`__MaxLocation`** (*data*, *field*)

This function returns the location of the maximum of a set of fields.

**`__ParticleSpinParameter`** (*data*)

This function returns the spin parameter for the baryons, but it uses the particles in calculating enclosed mass.

**`__TotalMass`** (*data*)

This function takes no arguments and returns the sum of cell masses and particle masses in the object.

**`__TotalQuantity`** (*data*, *fields*)

This function sums up a given field over the entire region

**Parameter** *fields* – The fields to sum up

**`__WeightedAverageQuantity`** (*data*, *field*, *weight*)

This function returns an averaged quantity.

**Parameters**

- *field* – The field to average

- *weight* – The field to weight by

### 12.2.3 `yt.lagos.FieldInfoContainer` Derived Field Objects

**class `FieldInfoContainer` ()**

This is a generic field container. It contains a list of potential derived fields, all of which know how to act on a data object and return a value. This object handles converting units as well as validating the availability of a given field.

**`add_field` (*name*, *function*=None, *\*\*kwargs*)**

Add a new field, along with supplemental metadata, to the list of available fields. This respects a number of arguments, all of which are passed on to the constructor for `DerivedField`.

**`keys` ()**

Return all the field names this object knows about.

**class `EnzoFieldContainer` ()**

This is a container for Enzo-specific fields.

**class `OrionFieldContainer` ()**

All Orion-specific fields are stored in here.

**class `DerivedField` (*name*, *function*, *convert\_function*=None, *units*="", *projected\_units*="", *take\_log*=True, *validators*=None, *particle\_type*=False, *vector\_field*=False, *display\_field*=True, *not\_in\_all*=False, *display\_name*=None, *projection\_conversion*='cm')**

This is the base class used to describe a cell-by-cell derived field.

#### Parameters

- *name* – is the name of the field.
- *function* – is a function handle that defines the field
- *convert\_function* – must convert to CGS, if it needs to be done
- *units* – is a mathtext-formatted string that describes the field
- *projected\_units* – if we display a projection, what should the units be?
- *take\_log* – describes whether the field should be logged
- *validators* – is a list of `FieldValidator` objects
- *particle\_type* – is this field based on particles?
- *vector\_field* – describes the dimensionality of the field
- *display\_field* – governs its appearance in the dropdowns in reason
- *not\_in\_all* – is used for baryon fields from the data that are not in all the grids
- *display\_name* – a name used in the plots
- *projection\_conversion* – which unit should we multiply by in a projection?

**`check_available` (*data*)**

This raises an exception of the appropriate type if the set of validation mechanisms are not met, and otherwise returns True.

**`get_dependencies` (*\*args*, *\*\*kwargs*)**

This returns a list of names of fields that this field depends on.

**`get_label` (*projected*=False)**

Return a data label for the given field, including units.

**get\_projected\_units()**

Return a string describing the units if the field has been projected.

**get\_source()**

Return a string containing the source of the function (if possible.)

**get\_units()**

Return a string describing the units.

**class ValidateParameter** (*parameters*)

This validator ensures that the parameter file has a given parameter.

**class ValidateDataField** (*field*)

This validator ensures that the output file has a given data field stored in it.

**class ValidateProperty** (*prop*)

This validator ensures that the data object has a given python attribute.

**class ValidateSpatial** (*ghost\_zones=0, fields=None*)

This validator ensures that the data handed to the field is of spatial nature – that is to say, 3-D.

**class ValidateGridType** ()

This validator ensures that the data handed to the field is an actual grid patch, not a covering grid of any kind.

## 12.2.4 yt.lagos.Profiles Profiling

Profiling in yt is a means of generating arbitrary histograms – for instance, phase diagrams, radial profiles, and even more complicated 3D examinations.

**class BinnedProfile1D** (*data\_source, n\_bins, bin\_field, lower\_bound, upper\_bound, log\_space=True, lazy\_reader=False, left\_collect=False*)

Bases: `yt.lagos.Profiles.BinnedProfile`

A ‘Profile’ produces either a weighted (or unweighted) average or a straight sum of a field in a bin defined by another field. In the case of a weighted average, we have:  $p_i = \text{sum}(w_i * v_i) / \text{sum}(w_i)$

We accept a *data\_source*, which will be binned into *n\_bins* by the field *bin\_field* between the *lower\_bound* and the *upper\_bound*. These bins may or may not be equally divided in *log\_space*, and the *lazy\_reader* flag controls whether we use a memory conservative approach.

**add\_fields** (*fields, weight='CellMassMsun', accumulation=False*)

We accept a list of *fields* which will be binned if *weight* is not None and otherwise summed. *accumulation* determines whether or not they will be accumulated from low to high along the appropriate axes.

**class BinnedProfile2D** (*data\_source, x\_n\_bins, x\_bin\_field, x\_lower\_bound, x\_upper\_bound, x\_log, y\_n\_bins, y\_bin\_field, y\_lower\_bound, y\_upper\_bound, y\_log, lazy\_reader=False, left\_collect=False*)

Bases: `yt.lagos.Profiles.BinnedProfile`

A ‘Profile’ produces either a weighted (or unweighted) average or a straight sum of a field in a bin defined by two other fields. In the case of a weighted average, we have:  $p_i = \text{sum}(w_i * v_i) / \text{sum}(w_i)$

We accept a *data\_source*, which will be binned into *x\_n\_bins* by the field *x\_bin\_field* between the *x\_lower\_bound* and the *x\_upper\_bound* and then again binned into *y\_n\_bins* by the field *y\_bin\_field* between the *y\_lower\_bound* and the *y\_upper\_bound*. These bins may or may not be equally divided in log-space as specified by *x\_log* and *y\_log*, and the *lazy\_reader* flag controls whether we use a memory conservative approach.

**add\_fields** (*fields, weight='CellMassMsun', accumulation=False*)

We accept a list of *fields* which will be binned if *weight* is not None and otherwise summed. *accumulation* determines whether or not they will be accumulated from low to high along the appropriate axes.



**write\_out** (*filename*, *format*='%0.16e')

Write out the values of x,y,v in ascii to *filename* for every field in the profile. Optionally a *format* can be specified.

**class BinnedProfile3D** (*data\_source*, *x\_n\_bins*, *x\_bin\_field*, *x\_lower\_bound*, *x\_upper\_bound*, *x\_log*, *y\_n\_bins*, *y\_bin\_field*, *y\_lower\_bound*, *y\_upper\_bound*, *y\_log*, *z\_n\_bins*, *z\_bin\_field*, *z\_lower\_bound*, *z\_upper\_bound*, *z\_log*, *lazy\_reader*=False)

Bases: `yt.lagos.Profiles.BinnedProfile`

**add\_fields** (*fields*, *weight*='CellMassMsun', *accumulation*=False)

We accept a list of *fields* which will be binned if *weight* is not None and otherwise summed. *accumulation* determines whether or not they will be accumulated from low to high along the appropriate axes.

**store\_profile** (*name*, *force*=False)

By identifying the profile with a fixed, user-input *name* we can store it in the serialized data section of the hierarchy file. *force* governs whether or not an existing profile with that name will be overwritten.

**class StoredBinnedProfile3D** (*pf*, *name*)

Bases: `yt.lagos.Profiles.BinnedProfile3D`

Given a *pf* parameterfile and the *name* of a stored profile, retrieve it into a read-only data structure.

## 12.2.5 yt.lagos.ContourFinder Contour Finding

Typically this is done via the `extract_connected_sets()` on a data object. However, you can call it manually, as is done in the clump finding scripts.

**identify\_contours** (*data\_source*, *field*, *min\_val*, *max\_val*, *cached\_fields*=None)

Given a *data\_source*, we will search for topologically connected sets in *field* between *min\_val* and *max\_val*.

**class GridConsiderationQueue** (*white\_list*, *priority\_func*=None)

This class exists to serve the contour finder. It ensures that we can create a cascading set of queue dependencies, and if a grid is touched again ahead of time we can bump it to the top of the queue again. It like has few uses.

## 12.2.6 yt.lagos.HaloFinding Halo Finding

yt now includes the HOP algorithm and implementation from the Enzo source distribution, with some modifications by both Stephen Skory and Matthew Turk.

**HaloFinder**

alias of `HOPHaloFinder`

**class HaloList** (*data\_source*, *dm\_only*=True)

Run hop on *data\_source* with a given density *threshold*. If *dm\_only* is set, only run it on the dark matter particles, otherwise on all particles. Returns an iterable collection of *HopGroup* items.

**write\_out** (*filename*)

Write out standard HOP information to *filename*.

**class Halo** (*halo\_list*, *id*, *indices*=None)

A data source that returns particle information about the members of a HOP-identified halo.

**bulk\_velocity** ()

Returns the mass-weighted average velocity.

**center\_of\_mass** ()

Calculate and return the center of mass.

**get\_sphere** (*center\_of\_mass=True*)

Returns an EnzoSphere centered on either the point of maximum density or the *center\_of\_mass*, with the maximum radius of the halo.

**maximum\_density** ()

Return the HOP-identified maximum density.

**maximum\_density\_location** ()

Return the location HOP identified as maximally dense.

**maximum\_radius** (*center\_of\_mass=True*)

Returns the maximum radius in the halo for all particles, either from the point of maximum density or from the (default) *center\_of\_mass*.

**total\_mass** ()

Returns the total mass in solar masses of the halo.

The specific halo finding algorithm can be specified by selecting the appropriate HaloFinder object. By default, HOP is used.

**class HOPHaloFinder** (*pf, threshold=160, dm\_only=True, padding=0.02*)

**class FOFHaloFinder** (*pf, link=0.20000000000000001, dm\_only=True, padding=0.02*)

## 12.3 yt . raven Plotting and Plot Interfaces

### 12.3.1 yt . raven . PlotCollection Plot Collection

PlotCollection is the basic means by which most of your backend-plotting will take place, and it contains a number of convenience functions for generating images and manipulating existing plots.

**class PlotCollection** (*pf, center=None, deliverator\_id=-1*)

Generate a collection of linked plots using *pf* as a source, optionally submitting to the deliverator with *deliverator\_id* and with *center*, which will otherwise be taken to be the point of maximum density.

**add\_cutting\_plane** (*field, normal, center=None, use\_colorbar=True, figure=None, axes=None, fig\_size=None, obj=None, \*\*kwargs*)

Generate a cutting plane of *field* with *normal*, centered at *center* (defaults to PlotCollection center) with *use\_colorbar* specifying whether the plot is naked or not and optionally providing pre-existing Matplotlib *figure* and *axes* objects. *fig\_size* in (height\_inches, width\_inches). If so desired, *obj* is a pre-existing cutting plane object.

**add\_phase\_object** (*data\_source, fields, cmap=None, weight='CellMassMsun', accumulation=False, x\_bins=64, x\_log=True, x\_bounds=None, y\_bins=64, y\_log=True, y\_bounds=None, lazy\_reader=False, id=None, axes=None, figure=None*)

Given a *data\_source*, and *fields*, automatically generate a 2D profile and plot it. *id* is used internally to add onto the prefix, and will be automatically generated if not given. Remainder of arguments are identical to [add\\_profile\\_object\(\)](#).

**add\_phase\_sphere** (*radius, unit, fields, \*\*kwargs*)

Given a *radius* and *unit*, generate a 2D profile from a sphere, with *fields* as the x,y,z. Automatically weights z by CellMassMsun. *kwargs* get passed onto [add\\_phase\\_object\(\)](#).

**add\_profile\_object** (*data\_source, fields, weight='CellMassMsun', accumulation=False, x\_bins=64, x\_log=True, x\_bounds=None, lazy\_reader=False, id=None, axes=None, figure=None*)

Use an existing data object, *data\_source*, to be the source of a one-dimensional profile. *fields* will define the x and y bin-by fields, *weight* is used to weight the y value, *accumulation* determines if y is summed along x, *x\_bins*, *x\_log* and *x\_bounds* define the means of choosing the bins. *id* is used internally to differentiate between multiple plots in a single collection. *lazy\_reader* determines the memory-conservative status.

**add\_profile\_sphere** (*radius*, *unit*, *fields*, *\*\*kwargs*)

Generate a spherical 1D profile, given only a *radius*, a *unit*, and at least two *fields*. Any remaining *kwargs* will be passed onto `add_profile_object()`.

**add\_projection** (*\*args*, *\*\*kwargs*)

Generate a projection of *field* along *axis*, optionally giving a *weight\_field*-weighted average with *use\_colorbar* specifying whether the plot is naked or not and optionally providing pre-existing Matplotlib *figure* and *axes* objects. *fig\_size* in (height\_inches, width\_inches)

**add\_projection\_interpolated** (*\*args*, *\*\*kwargs*)

Generate a projection of *field* along *axis*, optionally giving a *weight\_field*-weighted average with *use\_colorbar* specifying whether the plot is naked or not and optionally providing pre-existing Matplotlib *figure* and *axes* objects. *fig\_size* in (height\_inches, width\_inches)

The projection will be interpolated using the delaunay module, with natural neighbor interpolation.

**add\_slice** (*\*args*, *\*\*kwargs*)

Generate a slice through *field* along *axis*, optionally at [axis]=\*coord\*, with the *center* attribute given (some degeneracy with *coord*, but not complete), with *use\_colorbar* specifying whether the plot is naked or not and optionally providing pre-existing Matplotlib *figure* and *axes* objects. *fig\_size* in (height\_inches, width\_inches)

**add\_slice\_interpolated** (*\*args*, *\*\*kwargs*)

Generate a slice through *field* along *axis*, optionally at [axis]=\*coord\*, with the *center* attribute given (some degeneracy with *coord*, but not complete), with *use\_colorbar* specifying whether the plot is naked or not and optionally providing pre-existing Matplotlib *figure* and *axes* objects. *fig\_size* in (height\_inches, width\_inches)

The slice will be interpolated using the delaunay module, with natural neighbor interpolation.

**autoscale** ()

Turn back on autoscaling.

**clear\_plots** ()

Delete all plots and their attendant data.

**save** (*basename=None*, *format='png'*, *override=False*, *force\_save=False*)

Same plots with automatically generated names, prefixed with *basename* (including directory path) unless *override* is specified, and in *format*.

**set\_cmap** (*cmap*)

Change the colormap of all plots to *cmap*.

**set\_lim** (*lim*)

Shorthand for setting x,y at same time. *lim* should be formatted as (xmin,xmax,ymin,ymax)

**set\_width** (*width*, *unit*)

Set the width of the slices, cutting planes and projections to be *width units*

**set\_xlim** (*xmin*, *xmax*)

Set the x boundaries of all plots.

**set\_ylim** (*ymin*, *ymax*)

Set the y boundaries of all plots.

**set\_zlim** (*zmin*, *zmax*, *\*\*kwargs*)

Set the limits of the colorbar. 'min' or 'max' are possible inputs when combined with *dex=value*, where *value* gives the maximum number of dex to go above/below the min/max. If *value* is larger than the true range of values, min/max are limited to true range.

Only ONE of the following options can be specified. If all 3 are specified, they will be used in the following precedence order:

ticks - a list of floating point numbers at which to put ticks minmaxtick - display DEFAULT ticks with min & max also displayed nticks - if ticks not specified, can automatically determine a

number of ticks to be evenly spaced in log space

**switch\_field** (*field*)

Change all the fields displayed to be *field*

**switch\_z** (*field*)

Change all the fields displayed to be *field*

**class PlotCollectionInteractive** (\*args, \*\*kwargs)

**add\_cutting\_plane** (\*args, \*\*kwargs)

Generate a cutting plane of *field* with *normal*, centered at *center* (defaults to PlotCollection center) with *use\_colorbar* specifying whether the plot is naked or not and optionally providing pre-existing Matplotlib *figure* and *axes* objects. *fig\_size* in (height\_inches, width\_inches). If so desired, *obj* is a pre-existing cutting plane object.

**add\_phase\_object** (\*args, \*\*kwargs)

Given a *data\_source*, and *fields*, automatically generate a 2D profile and plot it. *id* is used internally to add onto the prefix, and will be automatically generated if not given. Remainder of arguments are identical to `add_profile_object()`.

**add\_phase\_sphere** (\*args, \*\*kwargs)

Given a *radius* and *unit*, generate a 2D profile from a sphere, with *fields* as the x,y,z. Automatically weights z by CellMassMsun. *kwargs* get passed onto `add_phase_object()`.

**add\_profile\_object** (\*args, \*\*kwargs)

Use an existing data object, *data\_source*, to be the source of a one-dimensional profile. *fields* will define the x and y bin-by fields, *weight* is used to weight the y value, *accumulation* determines if y is summed along x, *x\_bins*, *x\_log* and *x\_bounds* define the means of choosing the bins. *id* is used internally to differentiate between multiple plots in a single collection. *lazy\_reader* determines the memory-conservative status.

**add\_profile\_sphere** (\*args, \*\*kwargs)

Generate a spherical 1D profile, given only a *radius*, a *unit*, and at least two *fields*. Any remaining *kwargs* will be passed onto `add_profile_object()`.

**add\_projection** (\*args, \*\*kwargs)

Generate a projection of *field* along *axis*, optionally giving a *weight\_field*-weighted average with *use\_colorbar* specifying whether the plot is naked or not and optionally providing pre-existing Matplotlib *figure* and *axes* objects. *fig\_size* in (height\_inches, width\_inches)

**add\_projection\_interpolated** (\*args, \*\*kwargs)

Generate a projection of *field* along *axis*, optionally giving a *weight\_field*-weighted average with *use\_colorbar* specifying whether the plot is naked or not and optionally providing pre-existing Matplotlib *figure* and *axes* objects. *fig\_size* in (height\_inches, width\_inches)

The projection will be interpolated using the delaunay module, with natural neighbor interpolation.

**add\_slice** (\*args, \*\*kwargs)

Generate a slice through *field* along *axis*, optionally at [axis]=\*coord\*, with the *center* attribute given (some degeneracy with *coord*, but not complete), with *use\_colorbar* specifying whether the plot is naked or not and optionally providing pre-existing Matplotlib *figure* and *axes* objects. *fig\_size* in (height\_inches, width\_inches)

**add\_slice\_interpolated** (\*args, \*\*kwargs)

Generate a slice through *field* along *axis*, optionally at [axis]=\*coord\*, with the *center* attribute given (some degeneracy with *coord*, but not complete), with *use\_colorbar* specifying whether the plot is naked

or not and optionally providing pre-existing Matplotlib *figure* and *axes* objects. *fig\_size* in (height\_inches, width\_inches)

The slice will be interpolated using the delaunay module, with natural neighbor interpolation.

**autoscale** (\*args, \*\*kwargs)

Turn back on autoscaling.

**set\_cmap** (\*args, \*\*kwargs)

Change the colormap of all plots to *cmap*.

**set\_lim** (\*args, \*\*kwargs)

Shorthand for setting x,y at same time. *lim* should be formatted as (xmin,xmax,ymin,ymax)

**set\_width** (\*args, \*\*kwargs)

Set the width of the slices, cutting planes and projections to be *width units*

**set\_xlim** (\*args, \*\*kwargs)

Set the x boundaries of all plots.

**set\_ylim** (\*args, \*\*kwargs)

Set the y boundaries of all plots.

**set\_zlim** (\*args, \*\*kwargs)

Set the limits of the colorbar. 'min' or 'max' are possible inputs when combined with *dex=value*, where value gives the maximum number of dex to go above/below the min/max. If value is larger than the true range of values, min/max are limited to true range.

Only ONE of the following options can be specified. If all 3 are specified, they will be used in the following precedence order:

- ticks - a list of floating point numbers at which to put ticks
- minmaxtick - display DEFAULT ticks with min & max also displayed
- nticks - if ticks not specified, can automatically determine a number of ticks to be evenly spaced in log space

**switch\_field** (\*args, \*\*kwargs)

Change all the fields displayed to be *field*

**get\_multi\_plot** (nx, ny, colorbar='vertical', bw=4, dpi=300)

This returns *nx* and *ny* axes on a single figure, set up so that the *colorbar* can be placed either vertically or horizontally in a bonus column or row, respectively. The axes all have base width of *bw* inches.

## 12.3.2 yt.raven.PlotInterface Raw Plot Interface

**get\_slice** (pf, \*args, \*\*kwargs)

Get a single slice plot, with standard *field*, *axis* and *center* arguments.

**get\_projection** (pf, \*args, \*\*kwargs)

Get a single projection plot, with standard *field*, *axis* and *center* arguments.

## 12.3.3 yt.raven.FixedResolution Pixelization Interface

**class FixedResolutionBuffer** (data\_source, bounds, buff\_size, antialias=True)

Accepts a 2D data object, such as a Projection or Slice, and implements a protocol for generating a pixelized, fixed-resolution buffer. *bounds* is (px\_min,px\_max,py\_min,py\_max), *buff\_size* is (width, height), and *antialias* is a boolean referring to whether or not the buffer should have pixel boundary antialiasing.

**convert\_distance\_x** (distance)

This converts a real distance to a pixel distance in x.

**convert\_distance\_y** (*distance*)

This converts a real distance to a pixel distance in y.

**convert\_to\_pixel** (*coords*)

This converts a code-location to an image-location

**export\_fits** (*filename\_prefix*, *fields=None*)

This will export a set of FITS images of either the fields specified or all the fields already in the object. The output filenames are *filename\_prefix* plus an underscore plus the name of the field.

This requires the *pyfits* module, which is a standalone module provided by STSci to interface with FITS-format files.

**export\_hdf5** (*filename*, *fields=None*)

This function opens (append-mode) an HDF5 file and adds all of the requested *fields* (default: All) to the top level of the data file.

**open\_in\_ds9** (*field*, *take\_log=True*)

This will open a given field in DS9. This requires the *numdisplay* package, which is a simple download from STSci. Furthermore, it presupposed that it can connect to DS9 – that is, that DS9 is already open.

**class ObliqueFixedResolutionBuffer** (*data\_source*, *bounds*, *buff\_size*, *antialias=True*)

This object is a subclass of `yt.raven.FixedResolution.FixedResolutionBuffer` that supports non-aligned input data objects, primarily cutting planes.

Accepts a 2D data object, such as a Projection or Slice, and implements a protocol for generating a pixelized, fixed-resolution buffer. *bounds* is (px\_min,px\_max,py\_min,py\_max), *buff\_size* is (width, height), and *antialias* is a boolean referring to whether or not the buffer should have pixel boundary antialiasing.

**convert\_distance\_x** (*distance*)

This converts a real distance to a pixel distance in x.

**convert\_distance\_y** (*distance*)

This converts a real distance to a pixel distance in y.

**convert\_to\_pixel** (*coords*)

This converts a code-location to an image-location

**export\_fits** (*filename\_prefix*, *fields=None*)

This will export a set of FITS images of either the fields specified or all the fields already in the object. The output filenames are *filename\_prefix* plus an underscore plus the name of the field.

This requires the *pyfits* module, which is a standalone module provided by STSci to interface with FITS-format files.

**export\_hdf5** (*filename*, *fields=None*)

This function opens (append-mode) an HDF5 file and adds all of the requested *fields* (default: All) to the top level of the data file.

**open\_in\_ds9** (*field*, *take\_log=True*)

This will open a given field in DS9. This requires the *numdisplay* package, which is a simple download from STSci. Furthermore, it presupposed that it can connect to DS9 – that is, that DS9 is already open.

**class AnnuliProfiler** (*fixed\_buffer*, *center*, *num\_bins*, *min\_radius*, *max\_radius*)

This is a very simple class, principally used to sum up total values inside annuli in a fixed resolution buffer. It accepts *fixed\_buffer*, which should be a FixedResolutionBuffer, *center*, which is in pixel coordinates. *num\_bins*, *min\_radius* and *max\_radius* all refer to the binning properties for the annuli. Note that these are all in pixel values.

**sum** (*item*)

Returns the sum of a given field.



### 12.3.4 `yt.raven.Callbacks` Plot Modification Callbacks

These are all meant to be instantiated and fed into `yt.raven.RavenPlot.add_callback()`. For a more narrative discussion see [Plot Modification Mechanisms](#).

**class `ArrowCallback`** (*pos*, *code\_size*, *plot\_args=None*)

This adds an arrow pointing at *pos* with size *code\_size* in code units. *plot\_args* is a dict fed to matplotlib with arrow properties.

**class `ClumpContourCallback`** (*clumps*, *plot\_args=None*)

Take a list of *clumps* and plot them as a set of contours.

**class `ContourCallback`** (*field*, *ncont=5*, *factor=4*, *take\_log=False*, *clim=None*, *plot\_args=None*)

Add contours in *field* to the plot. *ncont* governs the number of contours generated, *factor* governs the number of points used in the interpolation, *take\_log* governs how it is contoured and *clim* gives the (upper, lower) limits for contouring.

**class `CoordAxesCallback`** (*unit=None*, *coords=False*)

Creates x and y axes for a VMPlot. In the future, it will attempt to guess the proper units to use.

**class `CuttingQuiverCallback`** (*field\_x*, *field\_y*, *factor*)

Get a quiver plot on top of a cutting plane, using *field\_x* and *field\_y*, skipping every *factor* datapoint in the discretization.

**class `GridBoundaryCallback`** (*alpha=1.0*, *min\_pix=1*)

Adds grid boundaries to a plot, optionally with *alpha*-blending. Cutoff for display is at *min\_pix* wide.

**class `HopCircleCallback`** (*hop\_output*, *max\_number=None*, *annotate=False*, *min\_size=20*, *max\_size=10000000*, *font\_size=8*, *print\_halo\_size=False*, *print\_halo\_mass=False*, *width=None*)

Accepts a `yt.lagos.HopList` *hop\_output* and plots up to *max\_number* (None for unlimited) halos as circles.

**class `HopParticleCallback`** (*hop\_output*, *p\_size=1.0*, *max\_number=None*, *min\_size=20*, *alpha=0.20000000000000001*)

Adds particle positions for the members of each halo as identified by HOP. Along *axis* up to *max\_number* groups in *hop\_output* that are larger than *min\_size* are plotted with *p\_size* pixels per particle; *alpha* determines the opacity of each particle.

**class `LabelCallback`** (*label*)

This adds a label to the plot.

**class `LinePlotCallback`** (*x*, *y*, *plot\_args=None*)

Over plot *x* and *y* with *plot\_args* fed into the plot.

**class `MarkerAnnotateCallback`** (*pos*, *marker='x'*, *plot\_args=None*)

Adds text *marker* at *pos* in code-arguments. *plot\_args* is a dict that will be forwarded to the plot command.

**class `NewParticleCallback`** (*width*, *p\_size=1.0*, *col='k'*, *stride=1.0*, *ptype=None*)

Adds particle positions, based on a thick slab along *axis* with a *width* along the line of sight. *p\_size* controls the number of pixels per particle, and *col* governs the color. *ptype* will restrict plotted particles to only those that are of a given type.

**class `ParticleCallback`** (*axis*, *width*, *p\_size=1.0*, *col='k'*, *stride=1.0*)

Adds particle positions, based on a thick slab along *axis* with a *width* along the line of sight. *p\_size* controls the number of pixels per particle, and *col* governs the color.

**class `PlotCallback`** (*\*args*, *\*\*kwargs*)

**class `PointAnnotateCallback`** (*pos*, *text*, *text\_args=None*)

This adds *text* at position *pos*, where *pos* is in code-space. *text\_args* is a dict fed to the text placement code.

**class QuiverCallback** (*field\_x, field\_y, factor*)

Adds a ‘quiver’ plot to any plot, using the *field\_x* and *field\_y* from the associated data, skipping every *factor* datapoints.

**class SphereCallback** (*center, radius, circle\_args=None, text=None, text\_args=None*)

A sphere centered at *center* in code units with radius *radius* in code units will be created, with optional *circle\_args*, *text*, and *text\_args*.

**class TextLabelCallback** (*pos, text, text\_args=None*)

Accepts a position in (0..1, 0..1) of the image, some text and optionally some text arguments.

**class TitleCallback** (*title='Plot'*)

Accepts a *title* and adds it to the plot

**class UnitBoundaryCallback** (*unit='au', factor=4, text\_annotate=True, text\_which=-2*)

Add on a plot indicating where *factor\*s of unit* are shown. Optionally *text\_annotate* on the *text\_which*-indexed box on display.

**class VelocityCallback** (*factor=16*)

Adds a ‘quiver’ plot of velocity to the plot, skipping all but every *factor* datapoint

**class VobozCircleCallback** (*voboz\_output, max\_number=None, annotate=False, min\_size=20, font\_size=8, print\_halo\_size=False*)

## 12.4 yt . reason GUI Methods and Objects

When the GUI is onscreen, you can open up a shell to not only interact with the data already in existence but to add new data objects. The one instance of ReasonMainWindow is known as *mainwindow* in the namespace of the interpreter.

Additionally, within the namespace of the Reason interpreter, you have access to *outputs*, which is a list of the outputs open in the main window, and *data\_objects*, which is every single derived data object generated in the GUI.

**class ReasonMainWindow** (*\*args, \*\*kwargs*)

## 12.5 Convenience Functions

These are functions that are meant to be used as a quick and easy mechanism for common operations.

### 12.5.1 yt . convenience Convenience Functions

**load** (*\*args, \*\*kwargs*)

This function attempts to determine the base data type of a filename or other set of arguments by calling `yt.lagos.StaticOutput._is_valid()` until it finds a match, at which point it returns an instance of the appropriate `yt.lagos.StaticOutput` subclass.

**all\_pfs** (*max\_depth=1, name\_spec='\*.hierarchy', \*\*kwargs*)

This function searches a directory and its sub-directories, up to a depth of *max\_depth*, for parameter files. It looks for the *name\_spec* and then instantiates an EnzoStaticOutput from each. All subsequent *kwargs* are passed on to the EnzoStaticOutput constructor.

**max\_spheres** (*width, unit, \*\*kwargs*)

This calls `all_pfs()` and then for each parameter file creates a `AMRSPHEREBase` for each one, centered on the point of highest density, with radius *width* in units of *unit*.



## 12.5.2 `yt.funcs` Miscellaneous Functions

### **iterable** (*obj*)

Grabbed from Python Cookbook / matplotlib.cbook. Returns true/false for *obj* iterable.

### **ensure\_list** (*obj*)

This function ensures that *obj* is a list. Typically used to convert a string to a list, for instance ensuring the *fields* as an argument is a list.

### **just\_one** (*obj*)

### **humanize\_time** (*secs*)

Takes *secs* and returns a nicely formatted string

### **time\_execution** (*func*)

Decorator for seeing how long a given function takes, depending on whether or not the global ‘yt.timefunctions’ config parameter is set.

This can be used like so:

```
@time_execution
```

```
def some_longrunning_function(...):
```

### **print\_tb** (*func*)

This function is used as a decorate on a function to have the calling stack printed whenever that function is entered.

This can be used like so:

```
@print_tb
```

```
def some_deeply_nested_function(...):
```

### **rootonly** (*func*)

This is a decorator that, when used, will only call the function on the root processor and then broadcast the results of the function to all other processors.

This can be used like so:

```
@rootonly
```

```
def some_root_only_function(...):
```

### **deprecate** (*func*)

This decorator issues a deprecation warning.

This can be used like so:

```
@rootonly
```

```
def some_really_old_function(...):
```

### **pdb\_run** (*func*)

This decorator inserts a pdb session on top of the call-stack into a function.

This can be used like so:

```
@rootonly
```

```
def some_function_to_debug(...):
```

**insert\_ipython** (*num\_up=1*)

Placed inside a function, this will insert an IPython interpreter at that current location. This will enabled detailed inspection of the current exeuction environment, as well as (optional) modification of that environment. *num\_up* refers to how many frames of the stack get stripped off, and defaults to 1 so that this function itself is stripped off.

**get\_pbar** (*title, maxval*)

This returns a progressbar of the most appropriate type, given a *title* and a *maxval*.

**only\_on\_root** (*func, \*args, \*\*kwargs*)

This function accepts a *func*, a set of *args* and *kwargs* and then only on the root processor calls the function. All other processors get “None” handed back.

**paste\_traceback** (*exc\_type, exc, tb*)

This is a traceback handler that knows how to paste to the pastebin. Should only be used in `sys.excepthook`.

### 12.5.3 yt.config Configuration System

**class YTConfigParser** (*fn, defaults=None*)

Bases: `ConfigParser.ConfigParser`

Simple class providing some functionality I wish existed in the ConfigParser module already

**add\_section** (*section*)

Create a new section in the configuration.

Raise DuplicateSectionError if a section by the specified name already exists. Raise ValueError if name is DEFAULT or any of it's case-insensitive variants.

**get** (*section, option, raw=False, vars=None*)

Get an option value for a given section.

All % interpolations are expanded in the return values, based on the defaults passed into the constructor, unless the optional argument ‘raw’ is true. Additional substitutions may be provided using the ‘vars’ argument, which must be a dictionary whose contents overrides any pre-existing defaults.

The section DEFAULT is special.

**has\_option** (*section, option*)

Check for the existence of a given option in a given section.

**has\_section** (*section*)

Indicate whether the named section is present in the configuration.

The DEFAULT section is not acknowledged.

**items** (*section, raw=False, vars=None*)

Return a list of tuples with (name, value) for each option in the section.

All % interpolations are expanded in the return values, based on the defaults passed into the constructor, unless the optional argument ‘raw’ is true. Additional substitutions may be provided using the ‘vars’ argument, which must be a dictionary whose contents overrides any pre-existing defaults.

The section DEFAULT is special.

**options** (*section*)

Return a list of option names for the given section name.

**read** (*filenames*)

Read and parse a filename or a list of filenames.

Files that cannot be opened are silently ignored; this is designed so that you can specify a list of potential configuration file locations (e.g. current directory, user's home directory, systemwide directory), and all existing configuration files in the list will be read. A single filename may also be given.

Return list of successfully read files.

**readfp** (*fp*, *filename=None*)

Like read() but the argument must be a file-like object.

The 'fp' argument must have a 'readline' method. Optional second argument is the 'filename', which if not given, is taken from fp.name. If fp has no 'name' attribute, '<??>' is used.

**remove\_option** (*section*, *option*)

Remove an option.

**remove\_section** (*section*)

Remove a file section.

**sections** ()

Return a list of section names, excluding [DEFAULT]

**set** (*section*, *opt*, *val*)

This sets an option named *opt* to *val* inside *section*, creating *section* if necessary.

**write** (*fp*)

Write an .ini-format representation of the configuration state.

## 12.6 yt.extensions Extensions API

There are some functions, routines and classes that utilize the yt API but aren't necessarily a part of the core functionality. These live inside the `yt/extensions/` subdirectory and are accessible by direct importation.

### 12.6.1 yt.extensions.coordinate\_transforms Coordinate Transforms

This module allows the user to regrid existing data into an arbitrary coordinate system. It comes with the machinery to automatically regrid onto spherical coordinates. *Module author: Matthew Turk <matthewturk@gmail.com>*

**spherical\_regrid** (*pf*, *nr*, *ntheta*, *nphi*, *rmax*, *fields*, *center=None*, *smoothed=True*)

This function takes a parameter file (*pf*) along with the *nr*, *ntheta* and *nphi* points to generate out to *rmax*, and it grids *fields* onto those points and returns a dict. *center* if supplied will be the center, otherwise the most dense point will be chosen. *smoothed* governs whether regular covering grids or smoothed covering grids will be used.

**arbitrary\_regrid** (*new\_grid*, *data\_source*, *fields*, *smoothed=True*)

This function accepts a dict of points 'x', 'y' and 'z' and a data source from which to interpolate new points, along with a list of fields it needs to regrid onto those xyz points. It then returns interpolated points. This has not been well-tested other than for regular spherical regridding.

### Sample Usage

```
from yt.mods import *
from yt.extensions.coordinate_transforms import *

pf = load("galaxy1200.dir/galaxy1200")

nr, ntheta, nphi = 128, 180, 180
```

```
fields = ["Density", "Temperature"]
center = [0.5, 0.5, 0.5]

regrid = spherical_regrid(pf, nr, ntheta, nphi, fields, center)
```

## 12.6.2 `yt.extensions.disk_analysis` Disk Analysis

The disk stacker is a mechanism for taking many oblique slices (see `cutting`) and stacking them together. The mechanism here is rather simple – you feed in a normal angle, a disk height, a width, and it makes many slices through the domain and adds them all up. This enables a ‘stacked’ image of the disk to be produced.

Once the object has been instantiated, you can access any field as per normal and it will stack it as requested:

*Module author: Matthew Turk <matthewturk@gmail.com>*

**class `StackedDiskImage`** (*pf, center, norm\_vec, thickness, width, nslices=100, buff\_size=(800, 800)*)

This class implements an AMR data object that will stack up oblique-slices to generate an image along an arbitrary axis.

## Sample Usage

```
from yt.mods import *
from yt.extensions.disk_analysis import StackedDiskImage

pf = load("galaxy1200.dir/galaxy1200")
norm_vec = [0.2, 0.4, 0.1]
center = [0.8, 0.5, 0.3]
thickness = 5.0 / pf['kpc']
width = 100.0 / pf['kpc']
n_slices = 200
image_size = (800, 800)

disk = StackedDiskImage(pf, center, norm_vec,
                        thickness, width, nslices, image_size)
my_disk_image = disk["Density"]
```

## 12.6.3 `yt.extensions.HaloProfiler` Halo Profiler

This module allows for systematic analysis and imaging of halos found in a simulation.

*Module author:*

*Britton Smith <brittonsmith@gmail.com>*

## 12.6.4 `yt.extensions.HierarchySubset` Hierarchy Subset

This module provides a mechanism for extracting a subset of a hierarchy. Typically this is used to export data to VTK or Amira format.

*Module author: Matthew Turk <matthewturk@gmail.com>*

**class `ConstructedRootGrid`** (*pf, level, left\_edge, right\_edge*)

This is a fake root grid, constructed by creating a `yt.lagos.CoveringGridBase` at a given *level* between *left\_edge* and *right\_edge*.

**class `ExtractedHierarchy`** (*pf, min\_level, max\_level=-1, offset=None, always\_copy=False*)

This is a class that extracts a hierarchy from another hierarchy, filling in regions as necessary. It accepts a

parameter file (*pf*), a *min\_level*, a *max\_level*, and alternately an *offset*. This class is typically or exclusively used to extract for the purposes of visualization.

### 12.6.5 `yt.extensions.SpectralIntegrator` Spectral Integrator

This module provides a mechanism for integrating emissivity output from CLOUDY and creating integrated X-ray emissivity fields. *Module author: Matthew Turk <matthewturk@gmail.com>*

**class `SpectralFrequencyIntegrator`** (*table, field\_names, bounds, ev\_bounds*)

From a table, interpolate over *field\_names* to get resultant luminosity. Table must be of the style such that it is ordered by [*field\_names*[0], *field\_names*[1], *ev*]

**add\_frequency\_bin\_field** (*ev\_min, ev\_max*)

Add a new field to the FieldInfoContainer, which is an integrated bin from *ev\_min* to *ev\_max*.

Returns the name of the new field.

**create\_table\_from\_textfiles** (*pattern, rho\_spec, e\_spec, T\_spec*)

This accepts a CLOUDY text file of emissivities and constructs an interpolation table for spectral integration.

### 12.6.6 `yt.extensions.lightcone` Light Cone Generation

`yt` has the facility to create light cones, which are stacks of images generated from a series of simulations. The code to generate this is in the module `lightcone`. *Module author: Britton Smith <brittonsmith@gmail.com>*

## 12.7 `yt.fido` File Storage and Management

In times past, this module was used for moving data and storing it in long-term storage. Now it is primarily used as a mechanism for loading parameter files without user intervention – in the case of object storage and serialization. (See *Storing and Loading Objects*.) There is still quite a bit of code that might be useful, but recent versions of Enzo have largely made it obsolete.

However, the `yt.fido.ParameterFileStore` is still quite useful for object serialization!

**class `ParameterFileStore`** (*in\_memory=False*)

This class is designed to be a semi-persistent storage for parameter files. By identifying each parameter file with a unique hash, objects can be stored independently of parameter files – when an object is loaded, the parameter file is as well, based on the hash. For storage concerns, only a few hundred will be retained in cache.

**check\_pf** (*pf*)

This will ensure that the parameter file (*pf*) handed to it is recorded in the storage unit. In doing so, it will update path and “last\_seen” information.

**flush\_db** ()

This flushes the storage to disk.

**get\_pf\_ctid** (*ctid*)

This returns a parameter file based on a CurrentTimeIdentifier.

**get\_pf\_hash** (*hash*)

This returns a parameter file based on a hash.

**init\_db** ()

This function ensures that the storage database exists and can be used.

**insert\_pf** (*pf*)

This will insert a new *pf* and flush the database to disk.

**read\_db()**

This will read the storage device from disk.

**wipe\_hash(hash)**

This removes a *hash* corresponding to a parameter file from the storage.

## 12.8 `yt.lagos.ParallelTools` Parallel Helper Functions

These functions are typically not used except in the construction of new parallel objects, but are documented here for future compatibility.

**class ParallelAnalysisInterface()**

This is an interface specification providing several useful utility functions for analyzing something in parallel.

**class ObjectIterator(pobj, just\_list=False, attr='\_grids')**

This is a generalized class that accepts a list of objects and then attempts to intelligently iterate over them.

**class ParallelObjectIterator(pobj, just\_list=False, attr='\_grids', round\_robin=False)**

This takes an object, *pobj*, that implements `ParallelAnalysisInterface`, and then does its thing, calling `initiaze` and `finalize` on the object.

**class ParallelDummy(name, bases, d)**

This is a base class that, on instantiation, replaces all attributes that don't start with `_` with `parallel_simple_proxy()`-wrapped attributes. Used as a metaclass.

**parallel\_simple\_proxy(func)**

This is a decorator that broadcasts the result of computation on a single processor to all other processors. To do so, it uses the `_processing` and `_distributed` flags in the object to check for blocks. Meant only to be used on objects that subclass `ParallelAnalysisInterface`.

**parallel\_passthrough(func)**

If we are not run in parallel, this function passes the input back as output; otherwise, the function gets called. Used as a decorator.

**parallel\_blocking\_call(func)**

This decorator blocks on entry and exit of a function.

**parallel\_splitter(f1, f2)**

This function returns either the function *f1* or *f2* depending on whether or not we're the root processor. Mainly used in class definitions.

**parallel\_root\_only(func)**

This decorator blocks and calls the function on the root processor, but does not broadcast results to the other processors.

# CHANGELOG

This is a non-comprehensive log of changes to the code.

## 13.1 Version 1.5

Version 1.5 features many new improvements, most prominently that of the addition of parallel computing abilities (see *Parallel Computation With YT*) and generalization for multiple AMR data formats, specifically both Enzo and Orion.

- Rewritten documentation
- Fully parallel slices, projections, cutting planes, profiles, quantities
- Parallel HOP
- Friends-of-friends halo finder
- Object storage and serialization
- Major performance improvements to the clump finder (factor of five)
- Generalized domain sizes
- Generalized field info containers
- Dark Matter-only simulations
- 1D and 2D simulations
- Better IO for HDF5 sets
- Support for the Orion AMR code
- Spherical re-gridding
- Halo profiler
- Disk image stacker
- Light cone generator
- Callback interface improved
- Several new callbacks
- New data objects – ortho and non-ortho rays, limited ray-tracing
- Fixed resolution buffers
- Spectral integrator for CLOUDY data

- Substantially better interactive interface
- Performance improvements *everywhere*
- Command-line interface to *many* common tasks
- Isolated plot handling, independent of PlotCollections

## 13.2 Version 1.0

- Initial release!



# INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*



# BIBLIOGRAPHY

- [vg06-kaehler] Kaehler, R., Wise, J., Abel, T., & Hege, H.-C. 2006, in Proceedings of the International Workshop on Volume Graphics 2006 (Boston: Eurographics / IEEE VGTC 2006), 103–110
- [2007ApJ-671-27H] Hallman, E. J., O’Shea, B. W., Burns, J. O., Norman, M. L., Harkness, R., & Wagner, R. 2007, ApJ, 671, 27
- [2009ApJ-696-96W] Wang, P. & Abel, T. 2009, ApJ, 696, 96
- [2009ApJ-691-441S] Smith, B. D., Turk, M. J., Sigurdsson, S., O’Shea, B. W., & Norman, M. L. 2009, ApJ, 691, 441
- [eishut98] Eisenstein, D. J. & Hut, P. 1998, ApJ, 498, 137



# MODULE INDEX

## Y

- `yt.config`, 148
- `yt.convenience`, 146
- `yt.extensions`, 149
  - `yt.extensions.coordinate_transforms`, 149
  - `yt.extensions.disk_analysis`, 150
  - `yt.extensions.HaloProfiler`, 150
  - `yt.extensions.HierarchySubset`, 150
  - `yt.extensions.lightcone`, 151
  - `yt.extensions.SpectralIntegrator`, 151
- `yt.fido`, 151
- `yt.funcs`, 147
- `yt.lagos`, 125
  - `yt.lagos.BaseDataTypes`, 131
  - `yt.lagos.ContourFinder`, 139
  - `yt.lagos.DerivedQuantities`, 136
  - `yt.lagos.FieldInfoContainer`, 137
  - `yt.lagos.HaloFinding`, 139
  - `yt.lagos.ParallelTools`, 152
  - `yt.lagos.Profiles`, 138
- `yt.raven`, 140
  - `yt.raven.Callbacks`, 145
  - `yt.raven.FixedResolution`, 143
  - `yt.raven.PlotCollection`, 140
  - `yt.raven.PlotInterface`, 143
- `yt.reason`, 146



# INDEX

## Symbols

`_AngularMomentumVector()` (in module `yt.lagos.DerivedQuantities`), 136  
`_BaryonSpinParameter()` (in module `yt.lagos.DerivedQuantities`), 136  
`_BulkVelocity()` (in module `yt.lagos.DerivedQuantities`), 136  
`_CenterOfMass()` (in module `yt.lagos.DerivedQuantities`), 136  
`_Extrema()` (in module `yt.lagos.DerivedQuantities`), 136  
`_IsBound()` (in module `yt.lagos.DerivedQuantities`), 136  
`_MaxLocation()` (in module `yt.lagos.DerivedQuantities`), 136  
`_ParticleSpinParameter()` (in module `yt.lagos.DerivedQuantities`), 136  
`_TotalMass()` (in module `yt.lagos.DerivedQuantities`), 136  
`_TotalQuantity()` (in module `yt.lagos.DerivedQuantities`), 136  
`_WeightedAverageQuantity()` (in module `yt.lagos.DerivedQuantities`), 136  
`__init__()` (built-in function), 85  
`_get_cut_mask()` (built-in function), 85  
`_get_list_of_grids()` (built-in function), 85  
`_is_fully_enclosed()` (built-in function), 85

## A

`add_cutting_plane()` (`yt.raven.PlotCollection` method), 140  
`add_cutting_plane()` (`yt.raven.PlotCollectionInteractive` method), 142  
`add_field()` (`yt.lagos.FieldInfoContainer` method), 137  
`add_fields()` (`yt.lagos.BinnedProfile1D` method), 138  
`add_fields()` (`yt.lagos.BinnedProfile2D` method), 138  
`add_fields()` (`yt.lagos.BinnedProfile3D` method), 139  
`add_frequency_bin_field()`  
(`yt.extensions.SpectralIntegrator.SpectralFrequencyIntegrator` method), 151  
`add_phase_object()` (`yt.raven.PlotCollection` method), 140  
`add_phase_object()` (`yt.raven.PlotCollectionInteractive` method), 142  
`add_phase_sphere()` (`yt.raven.PlotCollection` method), 140  
`add_phase_sphere()` (`yt.raven.PlotCollectionInteractive` method), 142  
`add_profile_object()` (`yt.raven.PlotCollection` method), 140  
`add_profile_object()` (`yt.raven.PlotCollectionInteractive` method), 142  
`add_profile_sphere()` (`yt.raven.PlotCollection` method), 141  
`add_profile_sphere()` (`yt.raven.PlotCollectionInteractive` method), 142  
`add_projection()` (`yt.raven.PlotCollection` method), 141  
`add_projection()` (`yt.raven.PlotCollectionInteractive` method), 142  
`add_projection_interpolated()` (`yt.raven.PlotCollection` method), 141  
`add_projection_interpolated()` (`yt.raven.PlotCollectionInteractive` method), 142  
`add_section()` (`yt.config.YTConfigParser` method), 148  
`add_slice()` (`yt.raven.PlotCollection` method), 141  
`add_slice()` (`yt.raven.PlotCollectionInteractive` method), 142  
`add_slice_interpolated()` (`yt.raven.PlotCollection` method), 141  
`add_slice_interpolated()` (`yt.raven.PlotCollectionInteractive` method), 142  
`all_pfs()` (in module `yt.convenience`), 146  
`AMR1DDData` (class in `yt.lagos`), 132  
`AMR2DDData` (class in `yt.lagos`), 132  
`AMR3DDData` (class in `yt.lagos`), 133  
`AMRCoveringGridBase` (class in `yt.lagos`), 135  
`AMRCuttingPlaneBase` (class in `yt.lagos`), 134  
`AMRCylinderBase` (class in `yt.lagos`), 135  
`AMRData` (class in `yt.lagos`), 132  
`AMRGridCollection` (class in `yt.lagos`), 135  
`AMRGridPatch` (class in `yt.lagos`), 130  
`AMRHierarchy` (class in `yt.lagos`), 128  
`AMROrthoRayBase` (class in `yt.lagos`), 134  
`AMRProjBase` (class in `yt.lagos`), 135  
`AMRRegionBase` (class in `yt.lagos`), 135

AMRSliceBase (class in yt.lagos), 134  
AMRSmoothedCoveringGridBase (class in yt.lagos), 135  
AMRSphereBase (class in yt.lagos), 135  
AnnuliProfiler (class in yt.raven.FixedResolution), 144  
arbitrary\_regrid() (in module  
yt.extensions.coordinate\_transforms), 149  
ArrowCallback (class in yt.raven.Callbacks), 145  
autoscale() (yt.raven.PlotCollection method), 141  
autoscale() (yt.raven.PlotCollectionInteractive method),  
143

## B

BinnedProfile1D (class in yt.lagos), 138  
BinnedProfile2D (class in yt.lagos), 138  
BinnedProfile3D (class in yt.lagos), 139  
bulk\_velocity() (yt.lagos.Halo method), 139

## C

center\_of\_mass() (yt.lagos.Halo method), 139  
check\_available() (yt.lagos.DerivedField method), 137  
check\_pf() (yt.fido.ParameterFileStore method), 151  
clear\_all() (yt.lagos.AMRGridPatch method), 130  
clear\_all() (yt.lagos.EnzoGridBase method), 129  
clear\_all() (yt.lagos.OrionGridBase method), 130  
clear\_all\_grid\_references() (yt.lagos.AMRGridPatch  
method), 131  
clear\_all\_grid\_references() (yt.lagos.EnzoGridBase  
method), 129  
clear\_all\_grid\_references() (yt.lagos.OrionGridBase  
method), 130  
clear\_data() (yt.lagos.AMR1DData method), 132  
clear\_data() (yt.lagos.AMR2DData method), 133  
clear\_data() (yt.lagos.AMR3DData method), 133  
clear\_data() (yt.lagos.AMRData method), 132  
clear\_data() (yt.lagos.AMRGridPatch method), 131  
clear\_data() (yt.lagos.EnzoGridBase method), 129  
clear\_data() (yt.lagos.OrionGridBase method), 130  
clear\_derived\_quantities() (yt.lagos.AMRGridPatch  
method), 131  
clear\_derived\_quantities() (yt.lagos.EnzoGridBase  
method), 129  
clear\_derived\_quantities() (yt.lagos.OrionGridBase  
method), 130  
clear\_plots() (yt.raven.PlotCollection method), 141  
ClumpContourCallback (class in yt.raven.Callbacks), 145  
ConstructedRootGrid (class in  
yt.extensions.HierarchySubset), 150  
ContourCallback (class in yt.raven.Callbacks), 145  
convert() (yt.lagos.AMR1DData method), 132  
convert() (yt.lagos.AMR2DData method), 133  
convert() (yt.lagos.AMR3DData method), 133  
convert() (yt.lagos.AMRData method), 132  
convert() (yt.lagos.AMRGridPatch method), 131  
convert() (yt.lagos.EnzoGridBase method), 129

convert() (yt.lagos.OrionGridBase method), 130  
convert\_distance\_x() (yt.raven.FixedResolution.FixedResolutionBuffer  
method), 143  
convert\_distance\_x() (yt.raven.FixedResolution.ObliqueFixedResolutionBu  
method), 144  
convert\_distance\_y() (yt.raven.FixedResolution.FixedResolutionBuffer  
method), 144  
convert\_distance\_y() (yt.raven.FixedResolution.ObliqueFixedResolutionBu  
method), 144  
convert\_to\_pixel() (yt.raven.FixedResolution.FixedResolutionBuffer  
method), 144  
convert\_to\_pixel() (yt.raven.FixedResolution.ObliqueFixedResolutionBuffe  
method), 144  
CoordAxesCallback (class in yt.raven.Callbacks), 145  
cosmology\_get\_units() (yt.lagos.EnzoStaticOutput  
method), 125  
create\_table\_from\_textfiles() (in module  
yt.extensions.SpectralIntegrator), 151  
cut\_region() (yt.lagos.AMR3DData method), 133  
CuttingQuiverCallback (class in yt.raven.Callbacks), 145

## D

deprecate() (in module yt.funcs), 147  
DerivedField (class in yt.lagos), 137

## E

ensure\_list() (in module yt.funcs), 147  
EnzoFieldContainer (class in yt.lagos), 137  
EnzoGridBase (class in yt.lagos), 129  
EnzoHierarchy (class in yt.lagos), 126  
EnzoStaticOutput (class in yt.lagos), 125  
export\_boxes\_pv() (yt.lagos.AMRHierarchy method),  
128  
export\_boxes\_pv() (yt.lagos.EnzoHierarchy method), 126  
export\_boxes\_pv() (yt.lagos.OrionHierarchy method),  
127  
export\_fits() (yt.raven.FixedResolution.FixedResolutionBuffer  
method), 144  
export\_fits() (yt.raven.FixedResolution.ObliqueFixedResolutionBuffer  
method), 144  
export\_hdf5() (yt.raven.FixedResolution.FixedResolutionBuffer  
method), 144  
export\_hdf5() (yt.raven.FixedResolution.ObliqueFixedResolutionBuffer  
method), 144  
export\_particles\_pb() (yt.lagos.AMRHierarchy method),  
128  
export\_particles\_pb() (yt.lagos.EnzoHierarchy method),  
126  
export\_particles\_pb() (yt.lagos.OrionHierarchy method),  
127  
extract\_connected\_sets() (yt.lagos.AMR3DData  
method), 133  
extract\_region() (yt.lagos.AMR3DData method), 133



ExtractedHierarchy (class  
yt.extensions.HierarchySubset), 150  
ExtractedRegionBase (class in yt.lagos), 135

## F

FakeGridForParticles (class in yt.lagos), 134  
FieldInfoContainer (class in yt.lagos), 137  
find\_max() (yt.lagos.AMRGridPatch method), 131  
find\_max() (yt.lagos.AMRHierarchy method), 128  
find\_max() (yt.lagos.EnzoGridBase method), 129  
find\_max() (yt.lagos.EnzoHierarchy method), 126  
find\_max() (yt.lagos.OrionGridBase method), 130  
find\_max() (yt.lagos.OrionHierarchy method), 127  
find\_min() (yt.lagos.AMRGridPatch method), 131  
find\_min() (yt.lagos.AMRHierarchy method), 128  
find\_min() (yt.lagos.EnzoGridBase method), 129  
find\_min() (yt.lagos.EnzoHierarchy method), 126  
find\_min() (yt.lagos.OrionGridBase method), 130  
find\_min() (yt.lagos.OrionHierarchy method), 127  
find\_point() (yt.lagos.AMRHierarchy method), 128  
find\_point() (yt.lagos.EnzoHierarchy method), 126  
find\_point() (yt.lagos.OrionHierarchy method), 127  
find\_ray\_grids() (yt.lagos.AMRHierarchy method), 128  
find\_ray\_grids() (yt.lagos.EnzoHierarchy method), 126  
find\_ray\_grids() (yt.lagos.OrionHierarchy method), 127  
find\_slice\_grids() (yt.lagos.AMRHierarchy method), 128  
find\_slice\_grids() (yt.lagos.EnzoHierarchy method), 126  
find\_slice\_grids() (yt.lagos.OrionHierarchy method), 127  
find\_sphere\_grids() (yt.lagos.AMRHierarchy method),  
128  
find\_sphere\_grids() (yt.lagos.EnzoHierarchy method),  
126  
find\_sphere\_grids() (yt.lagos.OrionHierarchy method),  
127  
findMax() (yt.lagos.AMRHierarchy method), 128  
findMax() (yt.lagos.EnzoHierarchy method), 126  
findMax() (yt.lagos.OrionHierarchy method), 127  
FixedResolutionBuffer (class  
yt.raven.FixedResolution), 143  
flush\_data() (yt.lagos.AMRCoveringGridBase method),  
135  
flush\_db() (yt.fido.ParameterFileStore method), 151  
FOFHaloFinder (class in yt.lagos), 140

## G

get() (yt.config.YTConfigParser method), 148  
get\_box\_grids() (yt.lagos.AMRHierarchy method), 128  
get\_box\_grids() (yt.lagos.EnzoHierarchy method), 126  
get\_box\_grids() (yt.lagos.OrionHierarchy method), 127  
get\_data() (yt.lagos.AMR2DData method), 133  
get\_data() (yt.lagos.AMRGridPatch method), 131  
get\_data() (yt.lagos.AMRHierarchy method), 128  
get\_data() (yt.lagos.EnzoGridBase method), 129  
get\_data() (yt.lagos.EnzoHierarchy method), 126

in get\_data() (yt.lagos.OrionGridBase method), 130  
get\_data() (yt.lagos.OrionHierarchy method), 127  
get\_dependencies() (yt.lagos.DerivedField method), 137  
get\_field\_parameter() (yt.lagos.AMR1DData method),  
132  
get\_field\_parameter() (yt.lagos.AMR2DData method),  
133  
get\_field\_parameter() (yt.lagos.AMR3DData method),  
133  
get\_field\_parameter() (yt.lagos.AMRData method), 132  
get\_field\_parameter() (yt.lagos.AMRGridPatch method),  
131  
get\_field\_parameter() (yt.lagos.EnzoGridBase method),  
129  
get\_field\_parameter() (yt.lagos.OrionGridBase method),  
130  
get\_global\_startindex() (yt.lagos.EnzoGridBase method),  
129  
get\_label() (yt.lagos.DerivedField method), 137  
get\_multi\_plot() (in module yt.raven), 143  
get\_parameter() (yt.lagos.EnzoStaticOutput method), 125  
get\_pbar() (in module yt.funcs), 148  
get\_pf\_ctid() (yt.fido.ParameterFileStore method), 151  
get\_pf\_hash() (yt.fido.ParameterFileStore method), 151  
get\_position() (yt.lagos.AMRGridPatch method), 131  
get\_position() (yt.lagos.EnzoGridBase method), 129  
get\_position() (yt.lagos.OrionGridBase method), 130  
get\_projected\_units() (yt.lagos.DerivedField method),  
137  
get\_projection() (in module yt.raven.PlotInterface), 143  
get\_slice() (in module yt.raven.PlotInterface), 143  
get\_smallest\_dx() (yt.lagos.AMRHierarchy method), 128  
get\_smallest\_dx() (yt.lagos.EnzoHierarchy method), 127  
get\_smallest\_dx() (yt.lagos.OrionHierarchy method), 127  
get\_source() (yt.lagos.DerivedField method), 138  
get\_sphere() (yt.lagos.Halo method), 139  
get\_units() (yt.lagos.DerivedField method), 138  
GridBoundaryCallback (class in yt.raven.Callbacks), 145  
GridConsiderationQueue (class in yt.lagos), 139

## H

Halo (class in yt.lagos), 139  
HaloFinder (in module yt.lagos), 139  
HaloList (class in yt.lagos), 139  
has\_field\_parameter() (yt.lagos.AMR1DData method),  
132  
has\_field\_parameter() (yt.lagos.AMR2DData method),  
133  
has\_field\_parameter() (yt.lagos.AMR3DData method),  
134  
has\_field\_parameter() (yt.lagos.AMRData method), 132  
has\_field\_parameter() (yt.lagos.AMRGridPatch method),  
131

`has_field_parameter()` (yt.lagos.EnzoGridBase method), 129

`has_field_parameter()` (yt.lagos.OrionGridBase method), 130

`has_key()` (yt.lagos.AMR1DDData method), 132

`has_key()` (yt.lagos.AMR2DDData method), 133

`has_key()` (yt.lagos.AMR3DDData method), 134

`has_key()` (yt.lagos.AMRData method), 132

`has_key()` (yt.lagos.AMRGridPatch method), 131

`has_key()` (yt.lagos.EnzoGridBase method), 129

`has_key()` (yt.lagos.EnzoStaticOutput method), 125

`has_key()` (yt.lagos.OrionGridBase method), 130

`has_key()` (yt.lagos.OrionStaticOutput method), 126

`has_key()` (yt.lagos.StaticOutput method), 126

`has_option()` (yt.config.YTConfigParser method), 148

`has_section()` (yt.config.YTConfigParser method), 148

`HopCircleCallback` (class in yt.raven.Callbacks), 145

`HOPHaloFinder` (class in yt.lagos), 140

`HopParticleCallback` (class in yt.raven.Callbacks), 145

`humanize_time()` (in module yt.funcs), 147

## I

`identify_contours()` (in module yt.lagos), 139

`init_db()` (yt.fido.ParameterFileStore method), 151

`insert_ipython()` (in module yt.funcs), 147

`insert_pf()` (yt.fido.ParameterFileStore method), 151

`interpolate_discretize()` (yt.lagos.AMR2DDData method), 133

`items()` (yt.config.YTConfigParser method), 148

`iterable()` (in module yt.funcs), 147

## J

`just_one()` (in module yt.funcs), 147

## K

`keys()` (yt.lagos.EnzoStaticOutput method), 125

`keys()` (yt.lagos.FieldInfoContainer method), 137

`keys()` (yt.lagos.OrionStaticOutput method), 126

`keys()` (yt.lagos.StaticOutput method), 126

## L

`LabelCallback` (class in yt.raven.Callbacks), 145

`LinePlotCallback` (class in yt.raven.Callbacks), 145

`load()` (in module yt.convenience), 146

`load_object()` (yt.lagos.AMRHierarchy method), 128

`load_object()` (yt.lagos.EnzoHierarchy method), 127

`load_object()` (yt.lagos.OrionHierarchy method), 127

## M

`MarkerAnnotateCallback` (class in yt.raven.Callbacks), 145

`max_spheres()` (in module yt.convenience), 146

`maximum_density()` (yt.lagos.Halo method), 140

`maximum_density_location()` (yt.lagos.Halo method), 140

`maximum_radius()` (yt.lagos.Halo method), 140

## N

`NewParticleCallback` (class in yt.raven.Callbacks), 145

## O

`ObjectIterator` (class in yt.lagos), 152

`ObliqueFixedResolutionBuffer` (class in yt.raven.FixedResolution), 144

`only_on_root()` (in module yt.funcs), 148

`open_in_ds9()` (yt.raven.FixedResolution.FixedResolutionBuffer method), 144

`open_in_ds9()` (yt.raven.FixedResolution.ObliqueFixedResolutionBuffer method), 144

`options()` (yt.config.YTConfigParser method), 148

`OrionFieldContainer` (class in yt.lagos), 137

`OrionGridBase` (class in yt.lagos), 130

`OrionHierarchy` (class in yt.lagos), 127

`OrionStaticOutput` (class in yt.lagos), 125

## P

`paint_grids()` (yt.lagos.AMR3DDData method), 134

`parallel_blocking_call()` (in module yt.lagos), 152

`parallel_passthrough()` (in module yt.lagos), 152

`parallel_root_only()` (in module yt.lagos), 152

`parallel_simple_proxy()` (in module yt.lagos), 152

`parallel_splitter()` (in module yt.lagos), 152

`ParallelAnalysisInterface` (class in yt.lagos), 152

`ParallelDummy` (class in yt.lagos), 152

`ParallelObjectIterator` (class in yt.lagos), 152

`ParameterFileStore` (class in yt.fido), 151

`ParticleCallback` (class in yt.raven.Callbacks), 145

`paste_traceback()` (in module yt.funcs), 148

`pdb_run()` (in module yt.funcs), 147

`PlotCallback` (class in yt.raven.Callbacks), 145

`PlotCollection` (class in yt.raven), 140

`PlotCollectionInteractive` (class in yt.raven), 142

`PointAnnotateCallback` (class in yt.raven.Callbacks), 145

`print_stats()` (yt.lagos.AMRHierarchy method), 129

`print_stats()` (yt.lagos.EnzoHierarchy method), 127

`print_stats()` (yt.lagos.OrionHierarchy method), 128

`print_tb()` (in module yt.funcs), 147

## Q

`QuiverCallback` (class in yt.raven.Callbacks), 145

## R

`read()` (yt.config.YTConfigParser method), 148

`read_db()` (yt.fido.ParameterFileStore method), 152

`readfp()` (yt.config.YTConfigParser method), 149

- readGlobalHeader() (yt.lagos.OrionHierarchy method), 128
- ReasonMainWindow (class in yt.reason), 146
- remove\_option() (yt.config.YTConfigParser method), 149
- remove\_section() (yt.config.YTConfigParser method), 149
- reslice() (yt.lagos.AMRSliceBase method), 134
- rootonly() (in module yt.funcs), 147
- ## S
- save() (yt.raven.PlotCollection method), 141
- save\_data() (yt.lagos.AMRHierarchy method), 129
- save\_data() (yt.lagos.EnzoHierarchy method), 127
- save\_data() (yt.lagos.OrionHierarchy method), 128
- save\_object() (yt.lagos.AMR1DData method), 132
- save\_object() (yt.lagos.AMR2DData method), 133
- save\_object() (yt.lagos.AMR3DData method), 134
- save\_object() (yt.lagos.AMRData method), 132
- save\_object() (yt.lagos.AMRGridPatch method), 131
- save\_object() (yt.lagos.AMRHierarchy method), 129
- save\_object() (yt.lagos.EnzoGridBase method), 130
- save\_object() (yt.lagos.EnzoHierarchy method), 127
- save\_object() (yt.lagos.OrionGridBase method), 130
- save\_object() (yt.lagos.OrionHierarchy method), 128
- sections() (yt.config.YTConfigParser method), 149
- select\_grids() (yt.lagos.AMR1DData method), 132
- select\_grids() (yt.lagos.AMR2DData method), 133
- select\_grids() (yt.lagos.AMR3DData method), 134
- select\_grids() (yt.lagos.AMRHierarchy method), 129
- select\_grids() (yt.lagos.EnzoHierarchy method), 127
- select\_grids() (yt.lagos.OrionHierarchy method), 128
- set() (yt.config.YTConfigParser method), 149
- set\_cmap() (yt.raven.PlotCollection method), 141
- set\_cmap() (yt.raven.PlotCollectionInteractive method), 143
- set\_field\_parameter() (yt.lagos.AMR1DData method), 132
- set\_field\_parameter() (yt.lagos.AMR2DData method), 133
- set\_field\_parameter() (yt.lagos.AMR3DData method), 134
- set\_field\_parameter() (yt.lagos.AMRData method), 132
- set\_field\_parameter() (yt.lagos.AMRGridPatch method), 131
- set\_field\_parameter() (yt.lagos.EnzoGridBase method), 130
- set\_field\_parameter() (yt.lagos.OrionGridBase method), 130
- set\_filename() (yt.lagos.EnzoGridBase method), 130
- set\_lim() (yt.raven.PlotCollection method), 141
- set\_lim() (yt.raven.PlotCollectionInteractive method), 143
- set\_width() (yt.raven.PlotCollectionInteractive method), 143
- set\_xlim() (yt.raven.PlotCollection method), 141
- set\_xlim() (yt.raven.PlotCollectionInteractive method), 143
- set\_ylim() (yt.raven.PlotCollection method), 141
- set\_ylim() (yt.raven.PlotCollectionInteractive method), 143
- set\_zlim() (yt.raven.PlotCollection method), 141
- set\_zlim() (yt.raven.PlotCollectionInteractive method), 143
- shift() (yt.lagos.AMRSliceBase method), 134
- SpectralFrequencyIntegrator (class in yt.extensions.SpectralIntegrator), 151
- SphereCallback (class in yt.raven.Callbacks), 146
- spherical\_regrid() (in module yt.extensions.coordinate\_transforms), 149
- StackedDiskImage (class in yt.extensions.disk\_analysis), 150
- StaticOutput (class in yt.lagos), 126
- store\_profile() (yt.lagos.BinnedProfile3D method), 139
- StoredBinnedProfile3D (class in yt.lagos), 139
- sum() (yt.raven.FixedResolution.AnnuliProfiler method), 144
- switch\_field() (yt.raven.PlotCollection method), 142
- switch\_field() (yt.raven.PlotCollectionInteractive method), 143
- switch\_z() (yt.raven.PlotCollection method), 142
- ## T
- TextLabelCallback (class in yt.raven.Callbacks), 146
- time\_execution() (in module yt.funcs), 147
- TitleCallback (class in yt.raven.Callbacks), 146
- total\_mass() (yt.lagos.Halo method), 140
- ## U
- UnitBoundaryCallback (class in yt.raven.Callbacks), 146
- ## V
- ValidateDataField (class in yt.lagos), 138
- ValidateGridType (class in yt.lagos), 138
- ValidateParameter (class in yt.lagos), 138
- ValidateProperty (class in yt.lagos), 138
- ValidateSpatial (class in yt.lagos), 138
- VelocityCallback (class in yt.raven.Callbacks), 146
- VobozCircleCallback (class in yt.raven.Callbacks), 146
- ## W
- wipe\_hash() (yt.fido.ParameterFileStore method), 152
- write() (yt.config.YTConfigParser method), 149
- write\_out() (yt.lagos.BinnedProfile2D method), 138
- write\_out() (yt.lagos.HaloList method), 139

## Y

- yt.config (module), 148
- yt.convenience (module), 146
- yt.extensions (module), 149
  - yt.extensions.coordinate\_transforms (module), 149
  - yt.extensions.disk\_analysis (module), 150
  - yt.extensions.HaloProfiler (module), 150
  - yt.extensions.HierarchySubset (module), 150
  - yt.extensions.lightcone (module), 151
  - yt.extensions.SpectralIntegrator (module), 151
- yt.fido (module), 151
- yt.funcs (module), 147
- yt.lagos (module), 125
  - yt.lagos.BaseDataTypes (module), 131
  - yt.lagos.ContourFinder (module), 139
  - yt.lagos.DerivedQuantities (module), 136
  - yt.lagos.FieldInfoContainer (module), 137
  - yt.lagos.HaloFinding (module), 139
  - yt.lagos.ParallelTools (module), 152
  - yt.lagos.Profiles (module), 138
- yt.raven (module), 140
  - yt.raven.Callbacks (module), 145
  - yt.raven.FixedResolution (module), 143
  - yt.raven.PlotCollection (module), 140
  - yt.raven.PlotInterface (module), 143
- yt.reason (module), 146
- YTConfigParser (class in yt.config), 148