06_Data_Objects_in_yt

September 26, 2014

1 Data Objects and Time Series Data

Just like before, we will load up yt. Since we'll be using pylab to plot some data in this notebook, we additionally tell matplotlib to place plots inline inside the notebook.

```
In []: %matplotlib inline
       import yt
       import numpy as np
       from matplotlib import pylab
       from yt.analysis_modules.halo_finding.api import HaloFinder
```

1.1 Time Series Data

Unlike before, instead of loading a single dataset, this time we'll load a bunch which we'll examine in sequence. This command creates a `DatasetSeries` object, which can be iterated over (including in parallel, which is outside the scope of this quickstart) and analyzed. There are some other helpful operations it can provide, but we'll stick to the basics here.

Note that you can specify either a list of filenames, or a glob (i.e., asterisk) pattern in this.

```
In []: ts = yt.DatasetSeries("enzo_tiny_cosmology/*/*.hierarchy")
```

1.1.1 Example 1: Simple Time Series

As a simple example of how we can use this functionality, let's find the min and max of the density as a function of time in this simulation. To do this we use the construction `for ds in ts` where `ds` means "Dataset" and `ts` is the "Time Series" we just loaded up. For each dataset, we'll create an object (`dd`) that covers the entire domain. (`all_data` is a shorthand function for this.) We'll then call the `extrema` Derived Quantity, and append the min and max to our extrema outputs.

```
In []: rho_ex = []
       times = []
       for ds in ts:
           dd = ds.all_data()
           rho_ex.append(dd.quantities.extrema("density"))
           times.append(ds.current_time.in_units("Gyr"))
       rho_ex = np.array(rho_ex)
```

Now we plot the minimum and the maximum:

```
In []: pylab.semilogy(times, rho_ex[:,0], '-xk', label='Minimum')
       pylab.semilogy(times, rho_ex[:,1], '-xr', label='Maximum')
       pylab.ylabel("Density ($g/cm^3$)")
       pylab.xlabel("Time (Gyr)")
       pylab.legend()
       pylab.ylim(1e-32, 1e-21)
       pylab.show()
```

1.1.2 Example 2: Advanced Time Series

Let's do something a bit different. Let's calculate the total mass inside halos and outside halos.

This actually touches a lot of different pieces of machinery in yt. For every dataset, we will run the halo finder HOP. Then, we calculate the total mass in the domain. Then, for each halo, we calculate the sum of the baryon mass in that halo. We'll keep running tallies of these two things.

```
In []: from yt.units import Msun

       mass = []
       zs = []
       for ds in ts:
           halos = HaloFinder(ds)
           dd = ds.all_data()
           total_mass = dd.quantities.total_quantity("cell_mass").in_units("Msun")
           total_in_baryons = 0.0*Msun
           for halo in halos:
               sp = halo.get_sphere()
               total_in_baryons += sp.quantities.total_quantity("cell_mass").in_units("Msun")
           mass.append(total_in_baryons/total_mass)
           zs.append(ds.current_redshift)
```

Now let's plot them!

```
In []: pylab.semilogx(zs, mass, '-xb')
       pylab.xlabel("Redshift")
       pylab.ylabel("Mass in halos / Total mass")
       pylab.xlim(max(zs), min(zs))
       pylab.ylim(-0.01, .18)
```

1.2 Data Objects

Time series data have many applications, but most of them rely on examining the underlying data in some way. Below, we'll see how to use and manipulate data objects.

1.2.1 Ray Queries

yt provides the ability to examine rays, or lines, through the domain. Note that these are not periodic, unlike most other data objects. We create a ray object and can then examine quantities of it. Rays have the special fields `t` and `dts`, which correspond to the time the ray enters a given cell and the distance it travels through that cell.

To create a ray, we specify the start and end points.

Note that we need to convert these arrays to numpy arrays due to a bug in matplotlib 1.3.1.

```
In []: ray = ds.ray([0.1, 0.2, 0.3], [0.9, 0.8, 0.7])
       pylab.semilogy(np.array(ray["t"]), np.array(ray["density"]))
```

```
In []: print ray["dts"]
```

```
In []: print ray["t"]
```

```
In []: print ray["x"]
```

1.2.2 Slice Queries

While slices are often used for visualization, they can be useful for other operations as well. yt regards slices as multi-resolution objects. They are an array of cells that are not all the same size; it only returns the cells at the highest resolution that it intersects. (This is true for all yt data objects.) Slices and projections have the special fields `px`, `py`, `pdx` and `pdy`, which correspond to the coordinates and half-widths in the pixel plane.

```
In []: ds = yt.load("IsolatedGalaxy/galaxy0030/galaxy0030")
       v, c = ds.find_max("density")
       sl = ds.slice(0, c[0])
       print sl["index", "x"]
       print sl["index", "z"]
       print sl["pdx"]
       print sl["gas", "density"].shape
```

If we want to do something interesting with a `Slice`, we can turn it into a `FixedResolutionBuffer`. This object can be queried and will return a 2D array of values.

```
In []: frb = sl.to_frb((50.0, 'kpc'), 1024)
       print frb["gas", "density"].shape
```

yt provides a few functions for writing arrays to disk, particularly in image form. Here we'll write out the log of `density`, and then use IPython to display it back here. Note that for the most part, you will probably want to use a `PlotWindow` for this, but in the case that it is useful you can directly manipulate the data.

```
In []: yt.write_image(np.log10(frb["gas", "density"]), "temp.png")
       from IPython.display import Image
       Image(filename = "temp.png")
```

1.2.3 Off-Axis Slices

yt provides not only slices, but off-axis slices that are sometimes called "cutting planes." These are specified by (in order) a normal vector and a center. Here we've set the normal vector to `[0.2, 0.3, 0.5]` and the center to be the point of maximum density.

We can then turn these directly into plot windows using `to_pw`. Note that the `to_pw` and `to_frb` methods are available on slices, off-axis slices, and projections, and can be used on any of them.

```
In []: cp = ds.cutting([0.2, 0.3, 0.5], "max")
       pw = cp.to_pw(fields = [("gas", "density")])
```

Once we have our plot window from our cutting plane, we can show it here.

```
In []: pw.show()
```

We can, as noted above, do the same with our slice:

```
In []: pws = sl.to_pw(fields=["density"])
       #pws.show()
       print pws.plots.keys()
```

1.2.4 Covering Grids

If we want to access a 3D array of data that spans multiple resolutions in our simulation, we can use a covering grid. This will return a 3D array of data, drawing from up to the resolution level specified when creating the data. For example, if you create a covering grid that spans two child grids of a single parent grid, it will fill those zones covered by a zone of a child grid with the data from that child grid. Where it is covered only by the parent grid, the cells from the parent grid will be duplicated (appropriately) to fill the covering grid.

There are two different types of covering grids: unsmoothed and smoothed. Smoothed grids will be filled through a cascading interpolation process; they will be filled at level 0, interpolated to level 1, filled at level 1, interpolated to level 2, filled at level 2, etc. This will help to reduce edge effects. Unsmoothed covering grids will not be interpolated, but rather values will be duplicated multiple times.

Here we create an unsmoothed covering grid at level 2, with the left edge at `[0.0, 0.0, 0.0]` and with dimensions equal to those that would cover the entire domain at level 2. We can then ask for the Density field, which will be a 3D array.

```
In []: cg = ds.covering_grid(2, [0.0, 0.0, 0.0], ds.domain_dimensions * 2**2)
       print cg["density"].shape
```

In this example, we do exactly the same thing: except we ask for a *smoothed* covering grid, which will reduce edge effects.

```
In []: scg = ds.smoothed_covering_grid(2, [0.0, 0.0, 0.0], ds.domain_dimensions * 2**2)
       print scg["density"].shape
```