07_Derived_Fields_in_yt

September 26, 2014

1 Derived Fields and Profiles

One of the most powerful features in yt is the ability to create derived fields that act and look exactly like fields that exist on disk. This means that they will be generated on demand and can be used anywhere a field that exists on disk would be used. Additionally, you can create them by just writing python functions.

```
In []: %matplotlib inline
       import yt
       import numpy as np
       from yt import derived_field
       from matplotlib import pylab
```

1.1 Derived Fields

This is an example of the simplest possible way to create a derived field. All derived fields are defined by a function and some metadata; that metadata can include units, LaTeX-friendly names, conversion factors, and so on. Fields can be defined in the way in the next cell. What this does is create a function which accepts two arguments and then provide the units for that field. In this case, our field is `dinosaurs` and our units are `K*cm/s`. The function itself can access any fields that are in the simulation, and it does so by requesting data from the object called `data`.

```
In []: @derived_field(name = "dinosaurs", units = "K * cm/s")
       def _dinos(field, data):
           return data["temperature"] * data["velocity_magnitude"]
```

One important thing to note is that derived fields must be defined *before* any datasets are loaded. Let's load up our data and take a look at some quantities.

```
In []: ds = yt.load("IsolatedGalaxy/galaxy0030/galaxy0030")
       dd = ds.all_data()
       print dd.quantities.keys()
```

One interesting question is, what are the minimum and maximum values of dinosaur production rates in our isolated galaxy? We can do that by examining the `extrema` quantity – the exact same way that we would for density, temperature, and so on.

```
In []: print dd.quantities.extrema("dinosaurs")
```

We can do the same for the average quantities as well.

```
In []: print dd.quantities.weighted_average_quantity("dinosaurs", weight="temperature")
```

1.2 A Few Other Quantities

We can ask other quantities of our data, as well. For instance, this sequence of operations will find the most dense point, center a sphere on it, calculate the bulk velocity of that sphere, calculate the baryonic angular momentum vector, and then the density extrema. All of this is done in a memory conservative way: if you have an absolutely enormous dataset, yt will split that dataset into pieces, apply intermediate reductions and then a final reduction to calculate your quantity.

```
In []: sp = ds.sphere("max", (10.0, 'kpc'))
       bv = sp.quantities.bulk_velocity()
       L = sp.quantities.angular_momentum_vector()
       rho_min, rho_max = sp.quantities.extrema("density")
       print bv
       print L
       print rho_min, rho_max
```

1.3 Profiles

yt provides the ability to bin in 1, 2 and 3 dimensions. This means discretizing in one or more dimensions of phase space (density, temperature, etc) and then calculating either the total value of a field in each bin or the average value of a field in each bin.

We do this using the objects `Profile1D`, `Profile2D`, and `Profile3D`. The first two are the most common since they are the easiest to visualize.

This first set of commands manually creates a profile object the sphere we created earlier, binned in 32 bins according to density between `rho_min` and `rho_max`, and then takes the density-weighted average of the fields `temperature` and (previously-defined) `dinosaurs`. We then plot it in a loglog plot.

```
In []: prof = yt.Profile1D(sp, "density", 32, rho_min, rho_max, True, weight_field="cell_mass")
       prof.add_fields(["temperature","dinosaurs"])
       pylab.loglog(np.array(prof.x), np.array(prof["temperature"]), "-x")
       pylab.xlabel('Density $(g/cm^3)$')
       pylab.ylabel('Temperature $(K)$')
```

Now we plot the `dinosaurs` field.

```
In []: pylab.loglog(np.array(prof.x), np.array(prof["dinosaurs"]), '-x')
       pylab.xlabel('Density $(g/cm^3)$')
       pylab.ylabel('Dinosaurs $(K cm / s)$')
```

If we want to see the total mass in every bin, we profile the `cell_mass` field with no weight. Specifying `weight=None` will simply take the total value in every bin and add that up.

```
In []: prof = yt.Profile1D(sp, "density", 32, rho_min, rho_max, True, weight_field=None)
       prof.add_fields(["cell_mass"])
       pylab.loglog(np.array(prof.x), np.array(prof["cell_mass"].in_units("Msun")), '-x')
       pylab.xlabel('Density $(g/cm^3)$')
       pylab.ylabel('Cell mass $(M_\odot)$')
```

In addition to the low-level `ProfileND` interface, it's also quite straightforward to quickly create plots of profiles using the `ProfilePlot` class. Let's redo the last plot using `ProfilePlot`

```
In []: prof = yt.ProfilePlot(sp, 'density', 'cell_mass', weight_field=None)
       prof.set_unit('cell_mass', 'Msun')
       prof.show()
```

1.4 Field Parameters

Field parameters are a method of passing information to derived fields. For instance, you might pass in information about a vector you want to use as a basis for a coordinate transformation. yt often uses things like `bulk_velocity` to identify velocities that should be subtracted off. Here we show how that works:

```
In []: sp_small = ds.sphere("max", (50.0, 'kpc'))
       bv = sp_small.quantities.bulk_velocity()

       sp = ds.sphere("max", (0.1, 'Mpc'))
       rv1 = sp.quantities.extrema("radial_velocity")

       sp.clear_data()
       sp.set_field_parameter("bulk_velocity", bv)
       rv2 = sp.quantities.extrema("radial_velocity")

       print bv
       print rv1
       print rv2
```