yt_attic Documentation

yt-project

Apr 18, 2020

Contents

This is a repository of yt analysis modules that have fallen by the wayside. Some have not been updated to stay current with yt development and some may not be working correctly. If you would like to revive anything in here and/or incorporate into your own software packages, please do! Also, consider listing your code on the yt Extensions page.

CHAPTER 1

Checking out the yt Attic

The yt Attic repository can be found here. In order to work with attic packages, yt must be installed. To download and install the attic, do the following:

```
$ git clone https://github.com/yt-project/yt_attic
$ cd yt_attic
$ python setup.py develop
```

CHAPTER 2

Importing Modules

Packages in the attic can be imported as `yt.extensions.attic.<package_name>`. For example,

```
from yt.extensions.attic.star_analysis.api import StarFormationRate
```

If you find something you like in here, give it a new home!

CHAPTER 3

In the Attic

Below is the existing documentation for all modules in the attic.

3.1 Enzo FoF Merger Tree

This is a merger tree code that once worked with yt's FoF and HoP halo finders. Unfortunately, its documentation has been lost to the sands of time, but the code can be found in the `yt_attic/enzofof_merger_tree` directory.

Warning: This is the state of the documentation before this module was moved to the attic. It is likely that code examples shown here do not work as advertised. If you would like to take on any code, you are welcome to its documentation.

3.2 Halo Mass Function

The Halo Mass Function extension is capable of outputting the halo mass function for a collection halos (input), and/or an analytical fit over a given mass range for a set of specified cosmological parameters. This extension is based on code generously provided by Brian O'Shea.

3.2.1 General Overview

A halo mass function can be created for the halos identified in a cosmological simulation, as well as analytic fits using any arbitrary set of cosmological parameters. In order to create a mass function for simulated halos, they must first be identified using a halo finder and loaded as a halo dataset object. The distribution of halo masses will then be found, and can be compared to the analytic prediction at the same redshift and using the same cosmological parameters as were used in the simulation. Care should be taken in this regard, as the analytic fit requires the specification of cosmological parameters that are not necessarily stored in the halo or simulation datasets, and must be specified by the user. Efforts have been made to set reasonable defaults for these parameters, but setting them to identically match those used in the simulation will produce a much better comparison.

Analytic halo mass functions can also be created without a halo dataset by providing either a simulation dataset or specifying cosmological parameters by hand. yt includes 5 analytic fits for the halo mass function which can be selected.

3.2.2 Analytical Fits

There are five analytical fits to choose from.

1. Press-Schechter (1974)
2. Jenkins (2001)
3. Sheth-Tormen (2002)
4. Warren (2006)
5. Tinker (2008)

We encourage reading each of the primary sources. In general, we recommend the Warren fitting function because it matches simulations over a wide range of masses very well. The Warren fitting function is the default (equivalent to not specifying `fitting_function` in `HaloMassFcn()`, below). The Tinker fit is for the $\Delta = 300$ fits given in the paper, which appears to fit HOP threshold=80.0 fairly well.

3.2.3 Basic Halo Mass Function Creation

The simplest way to create a halo mass function object is to simply pass it no arguments and let it use the default cosmological parameters.

```
from yt.analysis_modules.halo_mass_function.api import *

hmf = HaloMassFcn()
```

This will create a HaloMassFcn object off of which arrays holding the information about the analytic mass function hang. Creating the halo mass function for a set of simulated halos requires only the loaded halo dataset to be passed as an argument. This also creates the analytic mass function using all parameters that can be extracted from the halo dataset, at the same redshift, spanning a similar range of halo masses.

```
from yt.mods import *
from yt.analysis_modules.halo_mass_function.api import *

my_halos = load("rockstar_halos/halos_0.0.bin")
hmf = HaloMassFcn(halos_ds=my_halos)
```

A simulation dataset can be passed along with additional cosmological parameters to create an analytic mass function.

```
from yt.mods import *
from yt.analysis_modules.halo_mass_function.api import *

my_ds = load("RD0027/RedshiftOutput0027")
hmf = HaloMassFcn(simulation_ds=my_ds, omega_baryon0=0.05, primordial_index=0.96,
                  sigma8 = 0.8, log_mass_min=5, log_mass_max=9)
```

The analytic mass function can be created for a set of arbitrary cosmological parameters without any dataset being passed as an argument.

```
from yt.mods import *
from yt.analysis_modules.halo_mass_function.api import *

hmf = HaloMassFcn(omega_baryon0=0.05, omega_matter0=0.27,
                  omega_lambda0=0.73, hubble0=0.7, this_redshift=10,
                  log_mass_min=5, log_mass_max=9, fitting_function=5)
```

3.2.4 Keyword Arguments

- **simulation_ds** (*Simulation dataset object*) The loaded simulation dataset, used to set cosmological parameters. Default : None.
- **halos_ds** (*Halo dataset object*) The halos from a simulation to be used for creation of the halo mass function in the simulation. Default : None.
- **make_analytic** (*bool*) Whether or not to calculate the analytic mass function to go with the simulated halo mass function. Automatically set to true if a simulation dataset is provided. Default : True.
- **omega_matter0** (*float*) The fraction of the universe made up of matter (dark and baryonic). Default : 0.2726.
- **omega_lambda0** (*float*) The fraction of the universe made up of dark energy. Default : 0.7274.
- **omega_baryon0** (*float*) The fraction of the universe made up of baryonic matter. This is not always stored in the dataset and should be checked by hand. Default : 0.0456.
- **hubble0** (*float*) The expansion rate of the universe in units of 100 km/s/Mpc. Default : 0.704.
- **sigma8** (*float*) The amplitude of the linear power spectrum at z=0 as specified by the rms amplitude of mass-fluctuations in a top-hat sphere of radius 8 Mpc/h. This is not always stored in the dataset and should be checked by hand. Default : 0.86.
- **primoridal_index** (*float*) This is the index of the mass power spectrum before modification by the transfer function. A value of 1 corresponds to the scale-free primordial spectrum. This is not always stored in the dataset and should be checked by hand. Default : 1.0.
- **this_redshift** (*float*) The current redshift. Default : 0.
- **log_mass_min** (*float*) The log10 of the mass of the minimum of the halo mass range. This is set automatically by the range of halo masses if a simulated halo dataset is provided. If a halo dataset if not provided and no value is specified, it will be set to 5. Units: M_solar Default : None.
- **log_mass_max** (*float*) The log10 of the mass of the maximum of the halo mass range. This is set automatically by the range of halo masses if a simulated halo dataset is provided. If a halo dataset if not provided and no value is specified, it will be set to 16. Units: M_solar Default : None.
- **num_sigma_bins** (*float*) The number of bins (points) to use for the calculation of the analytic mass function. Default : 360.
- **fitting_function** (*int*) Which fitting function to use. 1 = Press-Schechter, 2 = Jenkins, 3 = Sheth-Tormen, 4 = Warren, 5 = Tinker Default : 4.

3.2.5 Outputs

A HaloMassFnc object has several arrays hanging off of it containing the

- **masses_sim**: Halo masses from simulated halos. Units: M_solar
- **n_cumulative_sim**: Number density of halos with mass greater than the corresponding mass in masses_sim. Units: comoving Mpc^-3

- **masses_analytic**: Masses used for the generation of the analytic mass function. Units: M_solar
- **n_cumulative_analytic**: Number density of halos with mass greater then the corresponding mass in masses_analytic. Units: comoving Mpc^-3
- **dndM_dM_analytic**: Differential number density of halos, (dn/dM)*dM.

After the mass function has been created for both simulated halos and the corresponding analytic fits, they can be plotted though something along the lines of

```
import yt
from yt.analysis_modules.halo_mass_function.api import *
import matplotlib.pyplot as plt

my_halos = yt.load("rockstar_halos/halos_0.0.bin")
hmf = HaloMassFcn(halos_ds=my_halos)

plt.loglog(hmf.masses_sim, hmf.n_cumulative_sim)
plt.loglog(hmf.masses_analytic, hmf.n_cumulative_analytic)
```

Attached to `hmf` is the convenience function `write_out`, which saves the halo mass function to a text file. (continued from above) .. code-block:: python

> hmf.write_out(prefix='hmf', analytic=True, simulated=True)

This writes the files `hmf-analytic.dat` with columns:

- mass [Msun]
- cumulative number density of halos [comoving Mpc^-3]
- (dn/dM)*dM (differential number density of halos) [comoving Mpc^-3]

and the file `hmf-simulated.dat` with columns:

- mass [Msun]
- cumulative number density of halos [comoving Mpc^-3]

Warning: This is the state of the documentation before this module was moved to the attic. It is likely that code examples shown here do not work as advertised. If you would like to take on any code, you are welcome to its documentation.

3.3 Star Particle Analysis

New in version 1.6.

This document describes tools in yt for analyzing star particles. The Star Formation Rate tool bins stars by time to produce star formation statistics over several metrics. A synthetic flux spectrum and a spectral energy density plot can be calculated with the Spectrum tool.

3.3.1 Star Formation Rate

This tool can calculate various star formation statistics binned over time. As input it can accept either a yt `data_source`, such as a region or sphere, or arrays containing the data for the stars you wish to analyze.

This example will analyze all the stars in the volume:

```
import yt
from yt.analysis_modules.star_analysis.api import StarFormationRate
ds = yt.load("Enzo_64/DD0030/data0030")
ad = ds.all_data()
sfr = StarFormationRate(ds, data_source=ad)
```

or just a small part of the volume i.e. a small sphere at the center of the simulation volume with radius 10% the box size:

```
import yt
from yt.analysis_modules.star_analysis.api import StarFormationRate
ds = yt.load("Enzo_64/DD0030/data0030")
sp = ds.sphere([0.5, 0.5, 0.5], 0.1)
sfr = StarFormationRate(ds, data_source=sp)
```

If the stars to be analyzed cannot be defined by a `data_source`, YTArrays can be passed. For backward compatibility it is also possible to pass generic numpy arrays. In this case, the units for the `star_mass` must be in $(\mathrm{M}_{\odot})$, the `star_creation_time` in code units, and the volume must be specified in (Mpc) as a float (but it doesn't have to be correct depending on which statistic is important).

```
import yt
from yt.analysis_modules.star_analysis.api import StarFormationRate
from yt.data_objects.particle_filters import add_particle_filter

def Stars(pfilter, data):
      return data[("all", "particle_type")] == 2
add_particle_filter("stars", function=Stars, filtered_type='all',
                    requires=["particle_type"])

ds = yt.load("enzo_tiny_cosmology/RD0009/RD0009")
ds.add_particle_filter('stars')
v, center = ds.find_max("density")
sp = ds.sphere(center, (50, "kpc"))

# This puts the particle data for *all* the particles in the sphere sp
# into the arrays sm and ct.
mass = sp[("stars", "particle_mass")].in_units('Msun')
age = sp[("stars", "age")].in_units('Myr')
ct = sp[("stars", "creation_time")].in_units('Myr')

# Pick out only old stars using Numpy array fancy indexing.
threshold = ds.quan(100.0, "Myr")
mass_old = mass[age > threshold]
ct_old = ct[age > threshold]

sfr = StarFormationRate(ds, star_mass=mass_old, star_creation_time=ct_old,
                        volume=sp.volume())
```

To output the data to a text file, use the command `.write_out`:

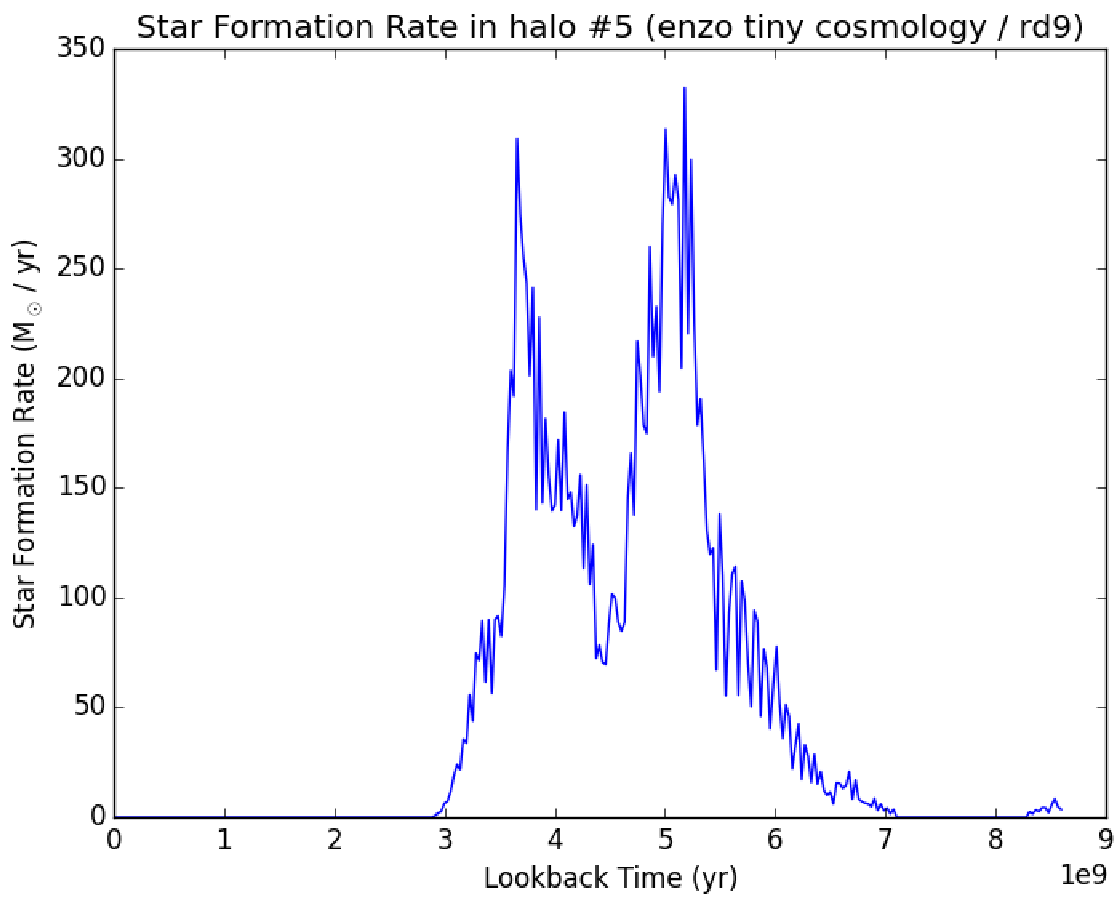
```
sfr.write_out(name="StarFormationRate.out")
```

In the file `StarFormationRate.out`, there are seven columns of data:

1. Time (yr)
2. Look-back time (yr)
3. Redshift

4. Star formation rate in this bin per year $(\mathrm{M}_\odot/\mathrm{yr})$
5. Star formation rate in this bin per year per Mpc**3 $(\mathrm{M}_\odot/\mathrm{h}/\mathrm{Mpc}^3)$
6. Stars formed in this time bin $(\mathrm{M}_\odot)$
7. Cumulative stars formed up to this time bin $(\mathrm{M}_\odot)$

The output is easily plotted. This is a plot for some test data (that may or may not correspond to anything physical) using columns #2 and #4 for the x and y axes, respectively:



It is possible to access the output of the analysis without writing to disk. Attached to the `sfr` object are the following arrays which are identical to the ones that are saved to the text file as above:

1. `sfr.time`
2. `sfr.lookback_time`
3. `sfr.redshift`
4. `sfr.Msol_yr`
5. `sfr.Msol_yr_vol`
6. `sfr.Msol`
7. `sfr.Msol_cumulative`

3.3.2 Synthetic Spectrum Generator

Based on code generously provided by Kentaro Nagamine <kn@physics.unlv.edu>, this will generate a synthetic spectrum for the stars using the publicly-available tables of Bruzual & Charlot (hereafter B&C). Please see their 2003 paper for more information and the main data distribution page for the original data. Based on the mass, age and metallicity of each star, a cumulative spectrum is generated and can be output in two ways, either raw, or as a spectral energy distribution.

This analysis toolkit reads in the B&C data from HDF5 files that have been converted from the original ASCII files (available at the link above). The HDF5 files are one-quarter the size of the ASCII files, and greatly reduce the time required to read the data off disk. The HDF5 files are available from the main yt website here. Both the Salpeter and Chabrier models have been converted, and it is simplest to download all the files to the same location. Please read the original B&C sources for information on the differences between the models.

In order to analyze stars, first the Bruzual & Charlot data tables need to be read in from disk. This is accomplished by initializing `SpectrumBuilder` and specifying the location of the HDF5 files with the `bcdir` parameter. The models are chosen with the `model` parameter, which is either *"chabrier"* or *"salpeter"*.

```
import yt
from yt.analysis_modules.star_analysis.api import SpectrumBuilder
ds = yt.load("enzo_tiny_cosmology/RD0009/RD0009")
spec = SpectrumBuilder(ds, bcdir="bc", model="chabrier")
```

In order to analyze a set of stars, use the `calculate_spectrum` command. It accepts either a `data_source`, or a set of YTarrays with the star information. Continuing from the above example:

```
v, center = ds.find_max("density")
sp = ds.sphere(center, (50, "kpc"))
spec.calculate_spectrum(data_source=sp)
```

If a subset of stars are desired, call it like this:

```
from yt.data_objects.particle_filters import add_particle_filter

def Stars(pfilter, data):
    return data[("all", "particle_type")] == 2
add_particle_filter("stars", function=Stars, filtered_type='all',
                    requires=["particle_type"])

# Pick out only old stars using Numpy array fancy indexing.
threshold = ds.quan(100.0, "Myr")
mass_old = sp[("stars", "age")][age > threshold]
metal_old = sp[("stars", "metallicity_fraction")][age > threshold]
ct_old = sp[("stars", "creation_time")][age > threshold]

spec.calculate_spectrum(star_mass=mass_old, star_creation_time=ct_old,
                        star_metallicity_fraction=metal_old)
```

For backward compatibility numpy arrays can be used instead for `star_mass` (in units $M_\odot$), `star_creation_time` and `star_metallicity_fraction` (in code units). Alternatively, when using either a `data_source` or individual arrays, the option `star_metallicity_constant` can be specified to force all the stars to have the same metallicity. If arrays are being used, the `star_metallicity_fraction` array need not be specified.

```
# Make all the stars have solar metallicity.
spec.calculate_spectrum(data_source=sp, star_metallicity_constant=0.02)
```

Newly formed stars are often shrouded by thick gas. With the `min_age` option of `calculate_spectrum`, young stars can be excluded from the spectrum. The units are in years. The default is zero, which is equivalent to including all stars.

```
spec.calculate_spectrum(data_source=sp, star_metallicity_constant=0.02,
                        min_age=ds.quan(1.0, "Myr"))
```

There are two ways to write out the data once the spectrum has been calculated. The command `write_out` outputs two columns of data:

1. Wavelength (Angstroms)
2. **Flux (Luminosity per unit wavelength** $(\mathrm{L}_\odot/$**Angstrom**$)$ **, where** $\mathrm{L}_\odot = 3.826 \cdot 10^{33}\,\mathrm{ergs/s}$).

and can be called simply, specifying the output file:

```
spec.write_out(name="spec.out")
```

The other way is to output a spectral energy density plot. Along with the `name` parameter, this command can also take the `flux_norm` option, which is the wavelength in Angstroms of the flux to normalize the distribution to. The default is 5200 Angstroms. This command outputs the data in two columns:

1. Wavelength (Angstroms)
2. Relative flux normalized to the flux at *flux_norm*.

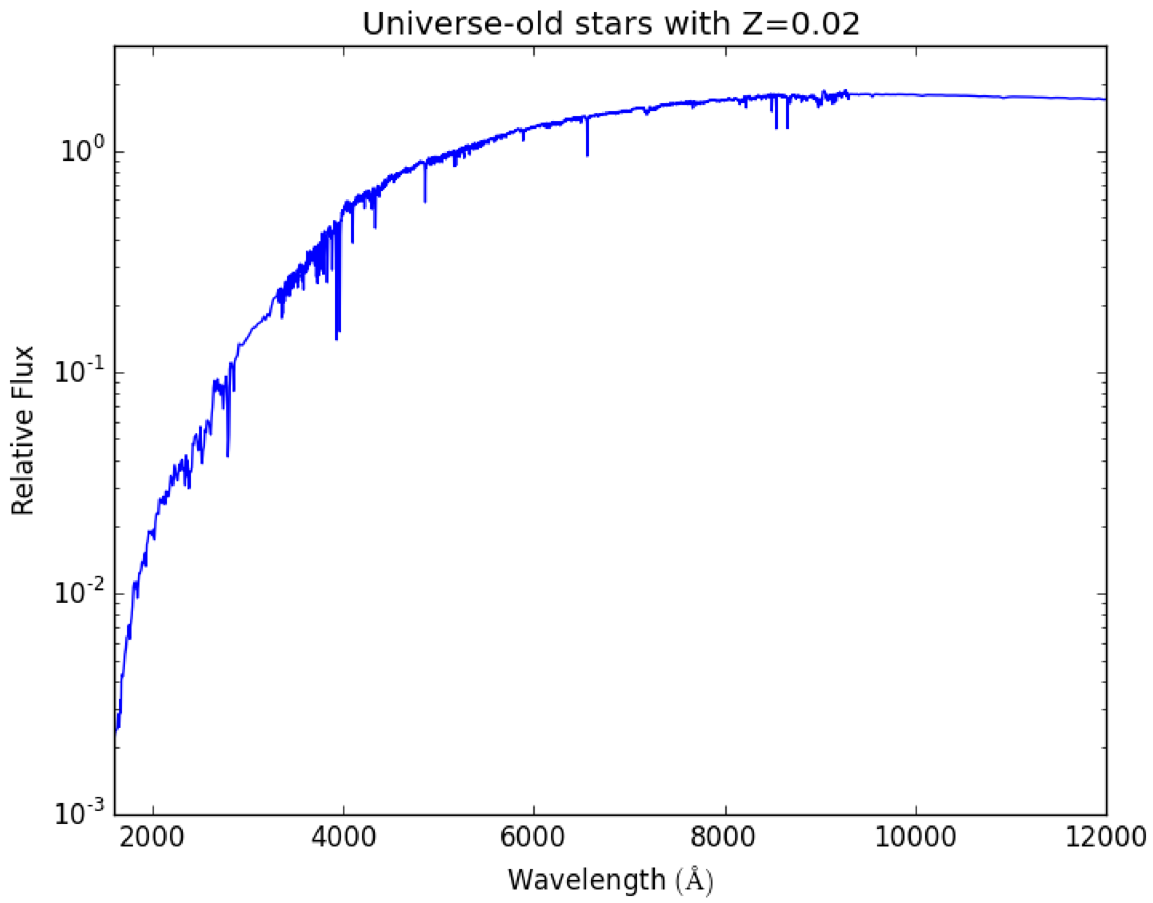
```
spec.write_out_SED(name="SED.out", flux_norm=5200)
```

Below is an example of an absurd SED for universe-old stars all with solar metallicity at a redshift of zero. Note that even in this example, a `ds` is required.

```
import yt
import numpy as np
from yt.analysis_modules.star_analysis.api import SpectrumBuilder

ds = yt.load("Enzo_64/DD0030/data0030")
spec = SpectrumBuilder(ds, bcdir="bc", model="chabrier")
sm = np.ones(100)
ct = np.zeros(100)
spec.calculate_spectrum(star_mass=sm, star_creation_time=ct,
                        star_metallicity_constant=0.02)
spec.write_out_SED('SED.out')
```

And the plot:



Iterate Over a Number of Halos

In this example below, the halos for a dataset are found, and the SED is calculated and written out for each.

```
import yt
from yt.analysis_modules.star_analysis.api import SpectrumBuilder
from yt.data_objects.particle_filters import add_particle_filter
from yt.analysis_modules.halo_finding.api import HaloFinder

def Stars(pfilter, data):
    return data[("all", "particle_type")] == 2
add_particle_filter("stars", function=Stars, filtered_type='all',
                    requires=["particle_type"])

ds = yt.load("enzo_tiny_cosmology/RD0009/RD0009")
ds.add_particle_filter('stars')
halos = HaloFinder(ds, dm_only=False)
# Set up the spectrum builder.
spec = SpectrumBuilder(ds, bcdir="bc", model="salpeter")

# Iterate over the halos.
for halo in halos:
    sp = halo.get_sphere()
```

(continues on next page)

(continued from previous page)

```
    spec.calculate_spectrum(
        star_mass=sp[("stars", "particle_mass")],
        star_creation_time=sp[("stars", "creation_time")],
        star_metallicity_fraction=sp[("stars", "metallicity_fraction")])
    # Write out the SED using the default flux normalization.
    spec.write_out_SED(name="halo%05d.out" % halo.id)
```

Warning: This is the state of the documentation before this module was moved to the attic. It is likely that code examples shown here do not work as advertised. If you would like to take on any code, you are welcome to its documentation.

3.4 Exporting to Sunrise

New in version 1.8.

Note: As of `yt-3.0`, the sunrise exporter is not currently functional. This functionality is still available in `yt-2.x`. If you would like to use these features in `yt-3.x`, help is needed to port them over. Contact the yt-users mailing list if you are interested in doing this.

The yt-Sunrise exporter essentially takes grid cell data and translates it into a binary octree format, attaches star particles, and saves the output to a FITS file Sunrise can read. For every cell, the gas mass, metals mass (a fraction of which is later assumed to be in the form of dust), and the temperature are saved. Star particles are defined entirely by their mass, position, metallicity, and a 'radius.' This guide outlines the steps to exporting the data, troubleshoots common problems, and reviews recommended sanity checks.

3.4.1 Simple Export

The code outlined here is a barebones Sunrise export:

```
from yt.mods import *
import numpy as na

ds = ARTDataset(file_amr)
potential_value,center=ds.find_min('Potential_New')
root_cells = ds.domain_dimensions[0]
le = np.floor(root_cells*center) #left edge
re = np.ceil(root_cells*center) #right edge
bounds = [(le[0], re[0]-le[0]), (le[1], re[1]-le[1]), (le[2], re[2]-le[2])]
#bounds are left edge plus a span
bounds = numpy.array(bounds,dtype='int')
amods.sunrise_export.export_to_sunrise(ds, out_fits_file,subregion_bounds = bounds)
```

To ensure that the camera is centered on the galaxy, we find the center by finding the minimum of the gravitational potential. The above code takes that center, and casts it in terms of which root cells should be extracted. At the moment, Sunrise accepts a strict octree, and you can only extract a 2x2x2 domain on the root grid, and not an arbitrary volume. See the optimization section later for workarounds. On my reasonably recent machine, the export process takes about 30 minutes.

Some codes do not yet enjoy full yt support. As a result, export_to_sunrise() can manually include particles in the yt output fits file:

```
import pyfits

col_list = []
col_list.append(pyfits.Column("ID", format="I", array=np.arange(mass_current.size)))
col_list.append(pyfits.Column("parent_ID", format="I", array=np.arange(mass_current.
↪size)))
col_list.append(pyfits.Column("position", format="3D", array=pos, unit="kpc"))
col_list.append(pyfits.Column("velocity", format="3D", array=vel, unit="kpc/yr"))
col_list.append(pyfits.Column("creation_mass", format="D", array=mass_initial, unit=
↪"Msun"))
col_list.append(pyfits.Column("formation_time", format="D", array=formation_time,␣
↪unit="yr"))
col_list.append(pyfits.Column("radius", format="D", array=star_radius, unit="kpc"))
col_list.append(pyfits.Column("mass", format="D", array=mass_current, unit="Msun"))
col_list.append(pyfits.Column("age_m", format="D", array=age, unit="yr"))
col_list.append(pyfits.Column("age_l", format="D", array=age, unit="yr"))
col_list.append(pyfits.Column("metallicity", format="D",array=z))
col_list.append(pyfits.Column("L_bol", format="D",array=np.zeros(mass_current.size)))
cols = pyfits.ColDefs(col_list)

amods.sunrise_export.export_to_sunrise(ds, out_fits_file,write_particles=cols,
    subregion_bounds = bounds)
```

This code snippet takes the stars in a region outlined by the `bounds` variable, organizes them into pyfits columns which are then passed to export_to_sunrise. Note that yt units are in CGS, and Sunrise accepts units in (physical) kpc, kelvin, solar masses, and years.

Remember that in Sunrise, photons are not spawned at the exact point of the star particle, but stochastically in a radius around it. Default to setting this radius to the resolution (or smoothing kernel) of your simulation - and then test that Sunrise is not sensitive to a doubling or halving of this number.

3.4.2 Sanity Check: Young Stars

Young stars are treated in a special way in Sunrise. Stars under 10 Myr do not emit in the normal fashion; instead they are replaced with MAPPINGS III particles that emulate the emission characteristics of star forming clusters. Among other things this involves a calculation of the local pressure, P/k, which Sunrise reports for debugging purposes and is something you should also check.

The code snippet below finds the location of every star under 10 Myr and looks up the cell containing it:

```
for x,a in enumerate(zip(pos,age)): #loop over stars
    center = x*ds['kpc']
    grid,idx = find_cell(ds.index.grids[0],center)
    pk[i] = grid['Pk'][idx]
```

This code is how Sunrise calculates the pressure, so we can add our own derived field:

```
def _Pk(field,data):
    #calculate pressure over Boltzmann's constant: P/k=(n/V)T
    #Local stellar ISM values are ~16500 Kcm^-3
    vol = data['cell_volume'].astype('float64')*data.ds['cm']**3.0 #volume in cm
    m_g = data["cell_mass"]*1.988435e33 #mass of H in g
    n_g = m_g*5.97e23 #number of H atoms
    teff = data["temperature"]
    val = (n_g/vol)*teff #should be of order 1e2-1e5
```

(continues on next page)

(continued from previous page)

```
    return  val
add_field("Pk", function=_Pk,units=r"Kcm^{-3}")
```

This snippet locates the cell containing a star and returns the grid and grid id.

```
def find_cell(grid,position):
    x=grid
    #print(grid.LeftEdge)
    for child in grid.Children:
        if numpy.all(child.LeftEdge  < position) and\
           numpy.all(child.RightEdge > position):
            return find_cell(child,position)

    #if the point is not contained within any of the child grids
    #find it within the extent of the current grid
    le,re = x.LeftEdge,x.RightEdge
    ad = x.ActiveDimensions
    span = (re-le)/ad
    idx = (position-le)/span
    idx = numpy.floor(idx)
    idx = numpy.int64(idx)
    assert numpy.all(idx < ad)
    return grid,idx
```

3.4.3 Sanity Check: Gas & Stars Line Up

If you add your star particles separately from the gas cell index, then it is worth checking that they still lined up once they've been loaded into Sunrise. This is fairly easy to do with a useful 'auxiliary' run. In Sunrise, set all of your rays to zero, (nrays_nonscatter, nrays_scatter,nrays_intensity,nrays_ir) except for nrays_aux, and this will produce an mcrx FITS file with a gas map, a metals map, a temperature*gass_mass map and a stellar map for each camera. As long as you keep some cameras at theta,phi = 0,0 or 90,0, etc., then a standard yt projection down the code's xyz axes should look identical:

```
pc.add_projection("density", 0, "density")
```

3.4.4 Convergence: High Resolution

At the moment, yt exports are the only grid data format Sunrise accepts. Otherwise, Sunrise typically inputs SPH particles or AREPO Voronoi grids. Among the many convergence checks you should perform is a high resolution check, which subdivides all leaves in the octree and copies the parent data into them, effectively increasing the resolution but otherwise not adding more information. Sunrise should yield similar results, and it is worth checking that indeed it does. Do so by just passing export_to_sunrise(. . . ,dummy_subdivide=True). The resulting file should be slightly less than 8 times larger because of newly added cells.

3.4.5 Other checks:

Check that the width of your extracted region is at least the size of your camera's field of view. It should probably be significantly larger than your FOV, and cutting that short could throw out otherwise interesting objects.

A good idea is to leverage yt to find the inertia tensor of the stars, find the rotation matrix that diagonalizes it, and use that to define cameras for Sunrise. Unless your code grid is aligned with your galaxy, this is required for getting edge-on or face-on shots.

3.4.6 The final product:

Above is a false color image where RGB are assigned to IR, optical and UV broadband filters, respectively.

Warning: This is the state of the documentation before this module was moved to the attic. It is likely that code examples shown here do not work as advertised. If you would like to take on any code, you are welcome to its documentation.

3.5 Two Point Functions

New in version 1.7.

Note: As of `yt-3.0`, the two point function analysis module is not currently functional. This functionality is still available in `yt-2.x`. If you would like to use these features in `yt-3.x`, help is needed to port them over. Contact the yt-users mailing list if you are interested in doing this.

The Two Point Functions framework (TPF) is capable of running several multi-dimensional two point functions simultaneously on a dataset using memory and workload parallelism. Examples of two point functions are structure functions and two-point correlation functions. It can analyze the entire simulation, or a small rectangular subvolume. The results can be output in convenient text format and in efficient HDF5 files.

3.5.1 Requirements

The TPF relies on the Fortran kD-tree that is used by the parallel HOP halo finder. The kD-tree is not built by default with yt so it must be built by hand.

3.5.2 Quick Example

It is very simple to setup and run a structure point function on a dataset. The script below will output the RMS velocity difference over the entire volume for a range of distances. There are some brief comments given below for each step.

```python
from yt.mods import *
from yt.analysis_modules.two_point_functions.api import *

ds = load("data0005")

# Calculate the S in RMS velocity difference between the two points.
# All functions have five inputs. The first two are containers
# for field values, and the second two are the raw point coordinates
# for the point pair. The fifth is the normal vector between the two points
# in r1 and r2. Not all the inputs must be used.
# The name of the function is used to name output files.
def rms_vel(a, b, r1, r2, vec):
    vdiff = a - b
    np.power(vdiff, 2.0, vdiff)
    vdiff = np.sum(vdiff, axis=1)
    return vdiff


# Initialize a function generator object.
# Set the input fields for the function(s),
# the number of pairs of points to calculate, how big a data queue to
# use, the range of pair separations and how many lengths to use,
# and how to divide that range (linear or log).
tpf = TwoPointFunctions(ds, ["velocity_x", "velocity_y", "velocity_z"],
    total_values=1e5, comm_size=10000,
    length_number=10, length_range=[1./128, .5],
    length_type="log")

# Adds the function to the generator. An output label is given,
# and whether or not to square-root the results in the text output is given.
# Note that the items below are being added as lists.
f1 = tpf.add_function(function=rms_vel, out_labels=['RMSvdiff'], sqrt=[True])

# Define the bins used to store the results of the function.
f1.set_pdf_params(bin_type='log', bin_range=[5e4, 5.5e13], bin_number=1000)

# Runs the functions.
tpf.run_generator()

# This calculates the M in RMS and writes out a text file with
# the RMS values and the lengths. The R happens because sqrt=True in
# add_function, above.
```

(continues on next page)

(continued from previous page)

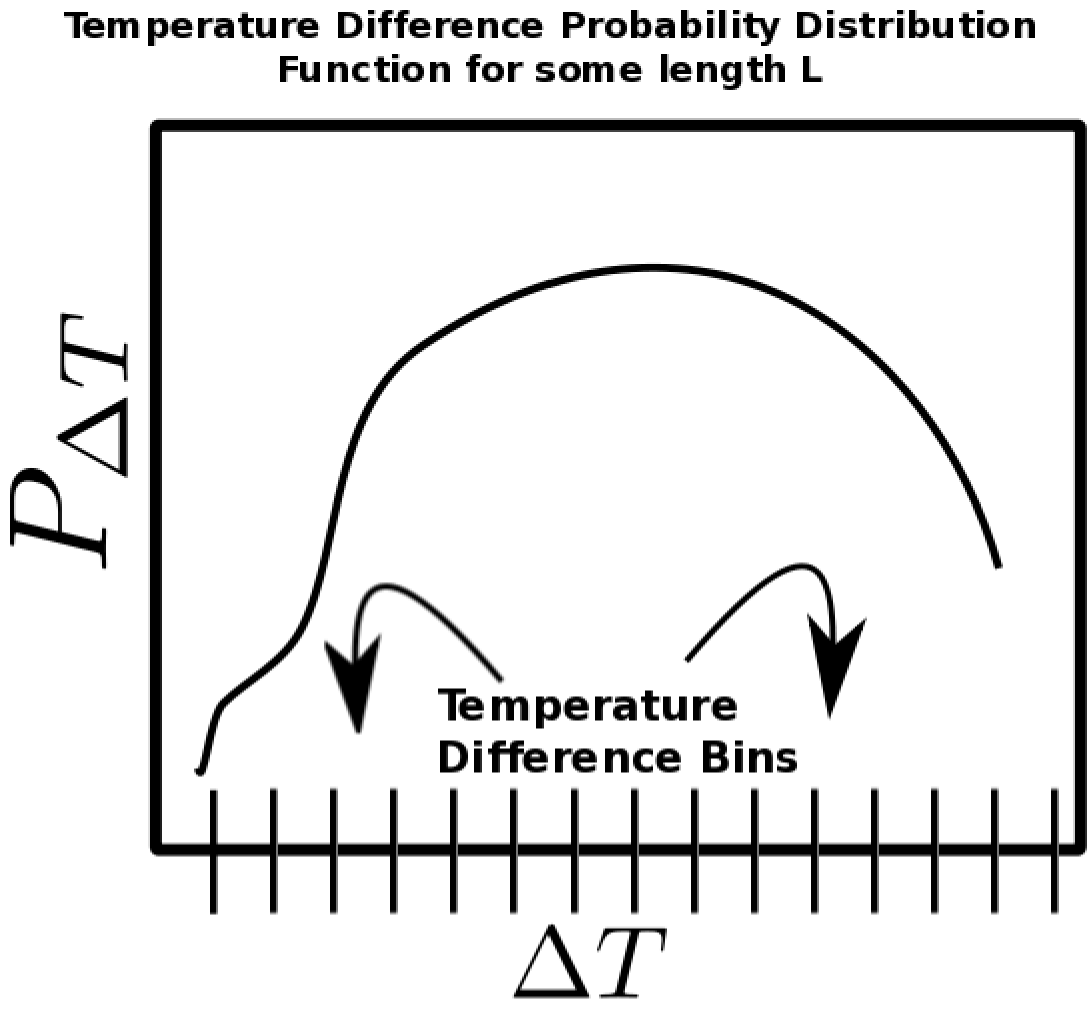
```
# If one is doing turbulence, the contents of this text file are what
# is wanted for plotting.
# The file is named 'rms_vel.txt'.
tpf.write_out_means()
# Writes out the raw PDF bins and bin edges to a HDF5 file.
# The file is named 'rms_vel.h5'.
tpf.write_out_arrays()
```

As an aside, note that any analysis function in yt can be accessed directly and imported automatically using the `amods` construct. Here is an abbreviated example:

```
from yt.mods import *
...
tpf = amods.two_point_functions.TwoPointFunctions(ds, ...)
```

3.5.3 Probability Distribution Function

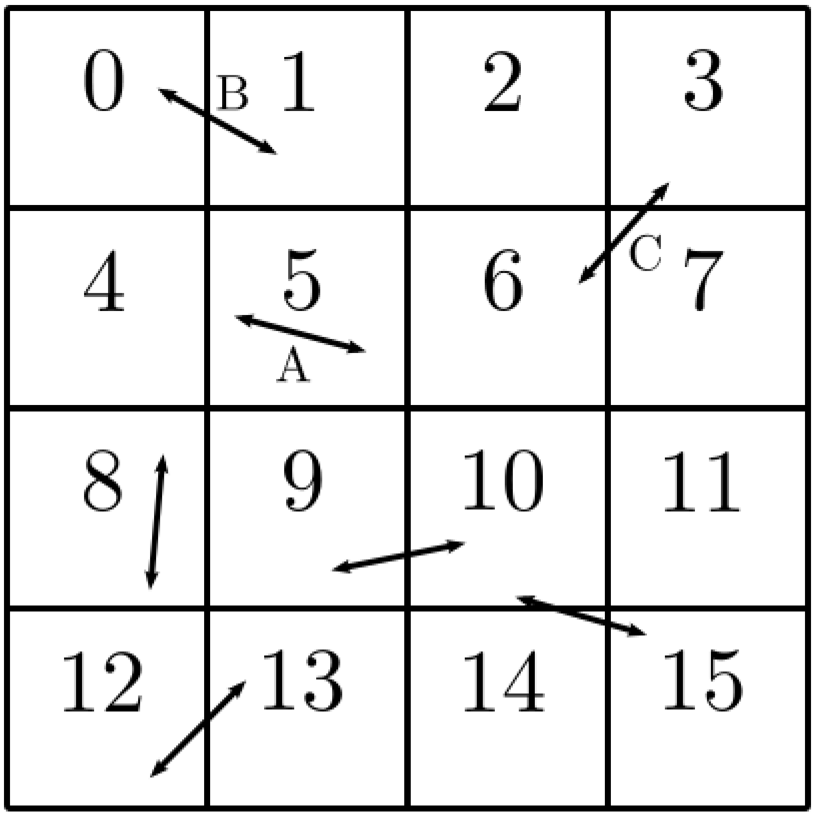
For a given length of separation between points, the TPF stores the Probability Distribution Function (PDF) of the output values. The PDF allows more varied analysis of the TPF output than storing the function itself. The image below assists in how to think about this. If the function is measuring the absolute difference in temperature between two points, for each point separation length L, the measured differences are binned by temperature difference (delta T). Therefore in the figure below, for a length L, the x-axis is temperature difference (delta T), and the y-axis is the probability of finding that temperature difference. To find the mean temperature difference for the length L, one just needs to multiply the value of the temperature difference bin by its probability, and add up over all the bins.


Temperature Difference Probability Distribution Function for some length L
$P_{\Delta T}$
Temperature Difference Bins
ΔT

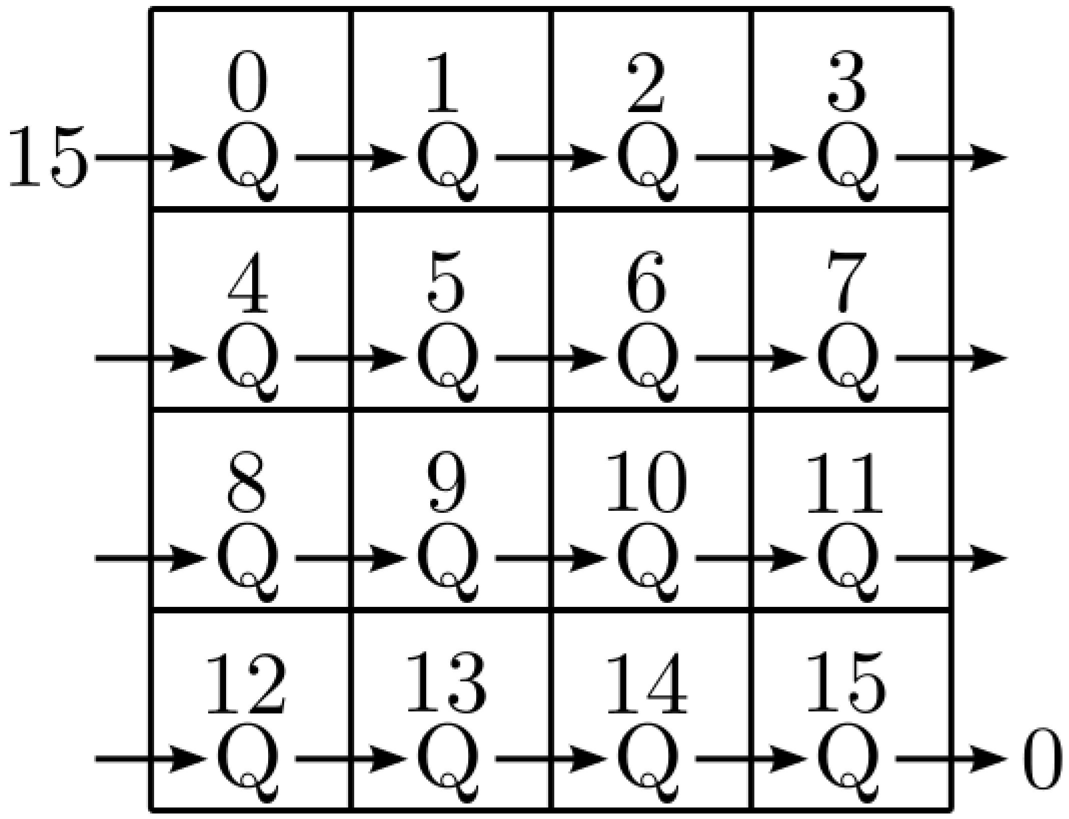
3.5.4 How It Works

In order to use the TPF, one must understand how it works. When run in parallel the defined analysis volume, whether it is the full volume or a small region, is subdivided evenly and each task is assigned a different subvolume. The total number of point pairs to be created per pair separation length is `total_values`, and each task is given an equal share of that total. Each task will create its share of `total_values` by first making a randomly placed point in its local volume. The second point will be placed a distance away with location set by random values of (phi, theta) in spherical coordinates and length by the length ranges. If that second point is inside the tasks subvolume, the functions are evaluated and their results binned. However, if the second point lies outside the subvolume (as in a different tasks subvolume), the point pair is stored in a point data queue, as well as the field values for the first point in a companion data queue. When a task makes its share of `total_values`, or it fills up its data queue with points it can't fully process, it passes its queues to its neighbor on the right. It then receives the data queues from its neighbor on the left, and processes the queues. If it can evaluate a point in the received data queues, meaning it can find the field values for the second point, it computes the functions for that point pair, and removes that entry from the queue. If it still needs to fulfill `total_values`, it can put its own point pair into that entry in the queues. Once the queues are full of points that a task cannot process, it passes them on. The data communication cycle ends when all tasks have made their share of `total_values`, and all the data queues are cleared. When all the cycles have run, the bins are added up globally to find the global PDF.

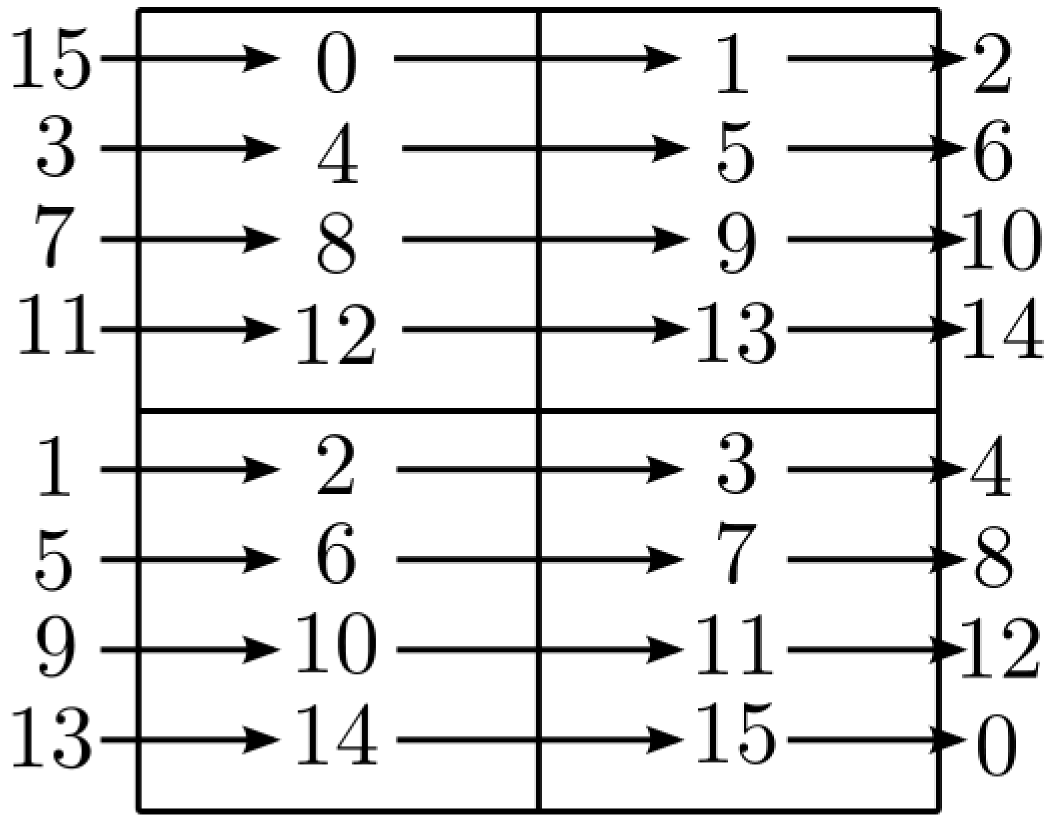
Below is a two-dimensional representation of how the full simulation is subdivided into 16 smaller subvolumes. Each subvolume is assigned to one of 16 tasks labelled with an integer [0-15]. Each task is responsible for only the field values inside its subvolume - it is completely ignorant about all the other subvolumes. When point separation rulers are laid down, some like the ruler labelled A, have both points completely inside a single subvolume. In this case, task 5 can evaluate the function(s) on its own. In situations like B or C, the points lie in different subvolumes, and no one task can evaluate the functions independently.



This next figure shows how the data queues are passed from task to task. Once task 0 is done with its points, or its queue is full, it passes the queue to task 1. Likewise, 1 passes to 2, and 15 passes back around to 0, completing the circle. If a point pair lies in the subvolumes of 0 and 15, it can take up to 15 communication cycles for that pair to be evaluated.



Sometimes the sizes of the data fields being computed on are not very large, and the memory-parallelism of the TPF isn't crucial. However, if one still wants to run with lots of processors to make large amounts of random pairs, subdividing the volumes as above is not as efficient as it could be due to communication overhead. By using the `vol_ratio` setting of TPF (see *Create the Function Generator Object*), the full volume can be subdivided into larger subvolumes than above, and tasks will own non-unique copies of the fields data. In the figure below, the two-dimensional volume has been subdivided into four subvolumes, and four tasks each own a copy of the data in each subvolume. As shown, the queues are handed off in the same order as before. But in this simple example, the maximum number of communication cycles for any point to be evaluated is three. This means that the communication overhead will be lower and runtimes somewhat faster.



3.5.5 A Step By Step Overview

In order to run the TPF, these steps must be taken:

1. Load yt (of course), and any other Python modules that are needed.
2. Define any non-default fields in the standard yt manner.
3. *Define Functions*.
4. *Create the Two Point Function Generator Object*.
5. *Add Functions*.
6. *Set PDF Parameters*.
7. *Run the TPF*.
8. *Output the Results*.

Define Functions

All functions must adhere to these specifications:

- There must be five input variables. The first two are arrays for the fields needed by the function, and the next two are the raw coordinate values for the points. The fifth input is an array with the normal vector between each of the points in r1 and r2.
- The output must be in array format.
- The names of the functions need to be unique.

The first two variables of a function are arrays that contain the field values. The order of the field values in the lists is set by the call to `TwoPointFunctions` (that comes later). In the example above, `a` and `b` contain the field velocities for the two points, respectively, in an N by M array, where N is equal to `comm_size` (set in `TwoPointFunctions`), and M is the total number of input fields used by functions. `a[:,0]` and `b[:,0]` are the `x-velocity` field values because that field is the first field given in the `TwoPointFunctions`.

The second two variables `r1` and `r2` are the raw point coordinates for the two points. The fifth input is an array containing the normal vector between each pair of points. These arrays are all N by 3 arrays. Note that they are not used in the example above because they are not needed.

Functions need to output in array format, with dimensionality N by R, where R is the dimensionality of the function. Multi-dimensional functions can be written that output several values simultaneously.

The names of the functions must be unique because they are used to name output files, and name collisions will result in over-written output.

Create the Two Point Function Generator Object

Before any functions can be added, the `TwoPointFunctions` object needs to be created. It has these inputs:

- `ds` (the only required input and is always the first term).
- Field list, required, an ordered list of field names used by the functions. The order in this list will need to be referenced when writing functions. Derived fields may be used here if they are defined first.
- `left_edge`, `right_edge`, three-element lists of floats: Used to define a sub-region of the full volume in which to run the TDS. Default=None, which is equivalent to running on the full volume. Both must be set to have any effect.
- `total_values`, integer: The number of random points to generate globally per point separation length. If run in parallel, each task generates its fair share of this number. Default=1000000.
- `comm_size`, integer: How many pairs of points that are stored in the data queue objects on each task. Too large wastes memory, and too small will result in longer run times due to extra communication cycles. Each unit of `comm_size` costs (6 + number_of_fields)*8 bytes, where number_of_fields is the size of the set of unique data fields used by all the functions added to the TPF. In the RMS velocity example above, number_of_fields=3, and a `comm_size` of 10,000 means each queue costs 10,000*8*(6+3) = 720 KB per task. Default=10000.
- `length_type`, string ("lin" or "log"): Sets how to evenly space the point separation lengths, either linearly or logarithmic (log10). Default="lin".
- `length_number`, integer: How many point separations to run. Default=10.
- `length_range`, two-element list of floats: Two values that define the minimum and maximum point separations to run over. The lengths that will be used are divided into `length_number` pieces evenly separated according to `length_type`. Default=None, which is equivalent to [sqrt(3)*dx, min_simulation_edge/2.], where min_simulation_edge is the length of the smallest edge (1D) of the simulation, and dx is the smallest cell size in the dataset. The sqrt(3) is there because that is the distance between opposite corners of a unit cube, and that guarantees that the point pairs will be in different cells for the most refined regions. If the first term of the list is -1, the minimum length will be automatically set to sqrt(3)*dx, ex: `length_range = [-1, 10/ds['kpc']]`.
- `vol_ratio`, integer: How to multiply-assign subvolumes to the parallel tasks. This number must be an integer factor of the total number of tasks or very bad things will happen. The default value of 1 will assign one task to each subvolume, and there will be an equal number of subvolumes as tasks. A value of 2 will assign two tasks to each subvolume and there will be one-half as many subvolumes as tasks. A value equal to the number of parallel tasks will result in each task owning a complete copy of all the fields data, meaning each task will be operating on the identical full volume. Setting this to -1 automatically adjusts `vol_ratio` such that all tasks are given the full volume.

- `salt`, integer: A number that will be added to the random number generator seed. Use this if a different random series of numbers is desired when keeping everything else constant from this set: (MPI task count, number of ruler lengths, ruler min/max, number of functions, number of point pairs per ruler length). Default: 0.
- `theta`, float: For random pairs of points, the second point is found by traversing a distance along a ray set by the angle (phi, theta) from the first point. To keep this angle constant, set `theta` to a value in the range [0, pi]. Default = None, which will randomize theta for every pair of points.
- `phi`, float: Similar to theta above, but the range of values is [0, 2*pi). Default = None, which will randomize phi for every pair of points.

Add Functions

Each function is added to the TPF using the `add_function` command. Each call must have the following inputs:

1. The function name as previously defined.
2. A list with label(s) for the output(s) of the function. Even if the function outputs only one value, this must be a list. These labels are used for output.
3. A list with bools of whether or not to sqrt the output, in the same order as the output label list. E.g. `[True, False]`.

The call to `add_function` returns a `FcnSet` object. For convenience, it is best to store the output in a variable (as in the example above) so it can be referenced later. The functions can also be referenced through the `TwoPointFunctions` object in the order in which they were added. So would `tpf[0]` refer to the same thing as `f1` in the quick example, above.

Set PDF Parameters

Once the function is added to the TPF, the probability distribution bins need to be defined for each using the command `set_pdf_params`. It has these inputs:

- `bin_type`, string or list of strings ("lin" or "log"): How to evenly subdivide the bins over the given range. If the function has multiple outputs, the input needs to be a list with equal elements.
- `bin_range`, list or list of lists: Define the min/max values for the bins for the output(s) of the function. If there are multiple outputs, there must be an equal number of lists.
- `bin_number`, integer or list of integers: How many bins to create over the min/max range defined by `bin_range` evenly spaced by the `bin_type` parameter. If there are multiple outputs, there must be an equal number of integers.

The memory costs associated with the PDF bins must be considered when writing an analysis script. There is one set of PDF bins created per function, per point separation length. Each PDF bin costs product(bin_number)*8 bytes, where product(bin_number) is the product of the entries in the bin_number list, and this is duplicated on every task. For multidimensional PDFs, the memory costs can grow very quickly. For example, for 3 functions, each with two outputs, with 1000 point separation lengths set for the TPF, and with 5000 PDF bins per output dimension, the PDF bins will cost: 3*1000*(5000)^2*8=600 GB of memory *per task*!

Note: `bin_number` actually specifies the number of *bin edges* to make, rather than the number of bins to make. The number of bins will actually be `bin_number`-1 because values are dropped into bins between the two closest bin edge values, and values outside the min/max bin edges are thrown away. If precisely `bin_number` bins are wanted, add 1 when setting the PDF parameters.

Run the TPF

The command `run_generator()` pulls the trigger and runs the TPF. There are no inputs.

After the generator runs, it will print messages like this, one per function:

```
yt          INFO       2010-03-13 12:46:54,541 Function rms_vel had 1 values too high␣
↪and 4960 too low that were not binned.
```

Consider changing the range of the PDF bins to reduce or eliminate un-binned values.

Output the Results

There are two ways to output data from the TPF for structure functions.

1. The command `write_out_means` writes out a text file per function that contains the means for each dimension of the function output for each point separation length. The file is named "function_name.txt", so in the example the file is named "rms_vel.txt". In the example above, the `sqrt=True` option is turned on, which square-roots the mean values. Here is some example output for the RMS velocity example:

```
# length     count        RMSvdiff
7.81250e-03 95040        8.00152e+04
1.24016e-02 100000       1.07115e+05
1.96863e-02 100000       1.53741e+05
3.12500e-02 100000       2.15070e+05
4.96063e-02 100000       2.97069e+05
7.87451e-02 99999        4.02917e+05
1.25000e-01 100000       5.54454e+05
1.98425e-01 100000       7.53650e+05
3.14980e-01 100000       9.57470e+05
5.00000e-01 100000       1.12415e+06
```

The `count` column lists the number of pair points successfully binned at that point separation length.

If the output is multidimensional, pass a list of bools to control the sqrt column by column (`sqrt=[False, True]`) to `add_function`. For multidimensional functions, the means are calculated by first collapsing the values in the PDF matrix in the other dimensions, before multiplying the result by the bin edges for that output dimension. So in the extremely simple fabricated case of:

```
# Temperature difference bin edges
# dimension 0
Tdiff_bins = [10, 100, 1000]
# Density difference bin edges
# dimension 1
Ddiff_bins = [50,500,5000]

# 2-D PDF for a point pair length of 0.05
PDF = [ [ 0.3, 0.1],
        [ 0.4, 0.2] ]
```

What the PDF is recording is that there is a 30% probability of getting a temperature difference between [10, 100), at the same time of getting a density difference between [50, 500). There is a 40% probability for Tdiff in [10, 100) and Ddiff in [500, 5000). The text output of this PDF is calculated like this:

```
# Temperature
T_PDF = PDF.sum(axis=0)
# ... which gets ...
```

(continues on next page)

(continued from previous page)

```
T_PDF = [0.7, 0.3]
# Then to get the mean, multiply by the centers of the temperature bins.
means = [0.7, 0.3] * [55, 550]
# ... which gets ...
means = [38.5, 165]
mean = sum(means)
# ... which gets ...
mean = 203.5

# Density
D_PDF = PDF.sum(axis=1)
# ... which gets ...
D_PDF = [0.4, 0.6]
# As above...
means = [0.4, 0.6] * [275, 2750]
mean = sum(means)
# ... which gets ...
mean = 1760
```

The text file would look something like this:

```
# length     count        Tdiff     Ddiff
0.05         980242       2.03500e+02 1.76000e+3
```

2. The command `write_out_arrays()` writes the raw PDF bins, as well as the bin edges for each output dimension to a HDF5 file named `function_name.h5`. Here is example content for the RMS velocity script above:

```
$ h5ls rms_vel.h5
bin_edges_00_RMSvdiff    Dataset {1000}
bin_edges_names          Dataset {1}
counts                   Dataset {10}
lengths                  Dataset {10}
prob_bins_00000          Dataset {999}
prob_bins_00001          Dataset {999}
prob_bins_00002          Dataset {999}
prob_bins_00003          Dataset {999}
prob_bins_00004          Dataset {999}
prob_bins_00005          Dataset {999}
prob_bins_00006          Dataset {999}
prob_bins_00007          Dataset {999}
prob_bins_00008          Dataset {999}
prob_bins_00009          Dataset {999}
```

Every HDF5 file produced will have the datasets `lengths`, `bin_edges_names`, and `counts`. `lengths` contains the list of the pair separation lengths used for the TPF, and is identical to the first column in the text output file. `bin_edges_names` lists the name(s) of the dataset(s) that contain the bin edge values. `counts` contains the number of successfully binned point pairs for each point separation length, and is equivalent to the second column in the text output file. In the HDF5 file above, the `lengths` dataset looks like this:

```
$ h5dump -d lengths rms_vel.h5
HDF5 "rms_vel.h5" {
DATASET "lengths" {
  DATATYPE  H5T_IEEE_F64LE
  DATASPACE  SIMPLE { ( 10 ) / ( 10 ) }
  DATA {
```

(continues on next page)

(continued from previous page)

```
  (0): 0.0078125, 0.0124016, 0.0196863, 0.03125, 0.0496063, 0.0787451,
  (6): 0.125, 0.198425, 0.31498, 0.5
  }
}
}
```

There are ten length values. `prob_bins_00000` is the PDF for pairs of points separated by the first length value given, which is 0.0078125. Points separated by 0.0124016 are recorded in `prob_bins_00001`, and so on. The entries in the `prob_bins` datasets are the raw PDF for that function for that point separation length. If the function has multiple outputs, the arrays stored in the datasets are multidimensional.

`bin_edges_names` looks like this:

```
$ h5dump -d bin_edges_names rms_vel.h5
HDF5 "rms_vel.h5" {
DATASET "bin_edges_names" {
  DATATYPE  H5T_STRING {
    STRSIZE 22;
    STRPAD H5T_STR_NULLPAD;
    CSET H5T_CSET_ASCII;
    CTYPE H5T_C_S1;
  }
  DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
  DATA {
  (0): "/bin_edges_00_RMSvdiff"
  }
}
}
```

This gives the names of the datasets that contain the bin edges, in the same order as the function output the data. If the function outputs several items, there will be more than one dataset listed in `bin_edges-names`. `bin_edges_00_RMSvdiff` therefore contains the (dimension 0) bin edges as specified when the PDF parameters were set. If there were other output fields, they would be named `bin_edges_01_outfield1`, `bin_edges_02_outfield2` respectively.

3.5.6 Strategies for Computational Efficiency

Here are a few recommendations that will make the function generator run as quickly as possible, in particular when running in parallel.

- Calculate how much memory the data fields and PDFs will require, and figure out what fraction can fit on a single compute node. For example (ignoring the PDF memory costs), if four data fields are required, and each takes up 8GB of memory (as in each field has 1e9 doubles), 32GB total is needed. If the analysis is being run on a machine with 4GB per node, at least eight nodes must be used (but in practice it is often just under 4GB available to applications, so more than eight nodes are needed). The number of nodes gives the minimal number of MPI tasks to use, which corresponds to the minimal volume decomposition required. Benchmark tests show that the function generator runs the quickest when each MPI task owns as much of the full volume as possible. If this number of MPI tasks calculated above is fewer than desired due to the number of pairs to be generated, instead of further subdividing the volume, use the `vol_ratio` parameter to multiply-assign tasks to the same subvolume. The total number of compute nodes will have to be increased because field data is being duplicated in memory, but tests have shown that things run faster in this mode. The bottom line: pick a vol_ratio that is as large as possible.
- The ideal `comm_size` appears to be around 1e5 or 1e6 in size.

- If possible, write the functions using only Numpy functions and methods. The input and output must be in array format, but the logic inside the function need not be. However, it will run much slower if optimized methods are not used.
- Run a few test runs before doing a large run so that the PDF parameters can be correctly set.

3.5.7 Advanced Two Point Function Techniques

Density Threshold

If points are to only be compared if they both are above some density threshold, simply pass the density field to the function, and return a value that lies outside the PDF min/max if the density is too low. Here are the modifications to the RMS velocity example to do this that requires a gas density of at least 1e-26 g cm^-3 at each point:

```
def rms_vel(a, b, r1, r2, vec):
  # Pick out points with only good densities
  a_good = a[:,3] >= 1.e-26
  b_good = b[:,3] >= 1.e-26
  # Pick out the pairs with both good densities
  both_good = np.bitwise_and(a_good, b_good)
  # Operate only on the velocity columns
  vdiff = a[:,0:3] - b[:,0:3]
  np.power(vdiff, 2.0, vdiff)
  vdiff = np.sum(vdiff, axis=1)
  # Multiplying by a boolean array has the effect of multiplying by 1 for
  # True, and 0 for False. This operation below will force pairs of not
  # good points to zero, outside the PDF (see below), and leave good
  # pairs unchanged.
  vdiff *= both_good
  return vdiff

...
tpf = TwoPointFunctions(ds, ["velocity_x", "velocity_y", "velocity_z", "density"],
    total_values=1e5, comm_size=10000,
    length_number=10, length_range=[1./128, .5],
    length_type="log")

tpf.add_function(rms_vel, ['RMSvdiff'], [False])
tpf[0].set_pdf_params(bin_type='log', bin_range=[5e4, 5.5e13], bin_number=1000)
```

Because 0 is outside of the `bin_range`, a pair of points that don't satisfy the density requirements do not contribute to the PDF. If density cutoffs are to be done in this fashion, the fractional volume that is above the density threshold should be calculated first, and `total_values` multiplied by the square of the inverse of this (which should be a multiplicative factor greater than one, meaning more point pairs will be generated to compensate for trashed points).

Multidimensional PDFs

It is easy to modify the example above to output in multiple dimensions. In this example, the ratio of the densities of the two points is recorded at the same time as the velocity differences.

```
from yt.mods import *
from yt.analysis_modules.two_point_functions.api import *

ds = load("data0005")
```

(continues on next page)

(continued from previous page)

```
# Calculate the S in RMS velocity difference between the two points.
# Also store the ratio of densities (keeping them >= 1).
# All functions have four inputs. The first two are containers
# for field values, and the second two are the raw point coordinates
# for the point pair. The name of the function is used to name
# output files.
def rms_vel_D(a, b, r1, r2, vec):
  # Operate only on the velocity columns
  vdiff = a[:,0:3] - b[:,0:3]
  np.power(vdiff, 2.0, vdiff)
  vdiff = np.sum(vdiff, axis=1)
  # Density ratio
  Dratio = np.max(a[:,3]/b[:,3], b[:,3]/a[:,3])
  return [vdiff, Dratio]

# Initialize a function generator object.
# Set the number of pairs of points to calculate, how big a data queue to
# use, the range of pair separations and how many lengths to use,
# and how to divide that range (linear or log).
tpf = TwoPointFunctions(ds, ["velocity_x", "velocity_y", "velocity_z", "density"],
    total_values=1e5, comm_size=10000,
    length_number=10, length_range=[1./128, .5],
    length_type="log")

# Adds the function to the generator.
f1 = tpf.add_function(rms_vel, ['RMSvdiff', 'Dratio'], [True, False])

# Define the bins used to store the results of the function.
# Note that the bin edges can have different division, "lin" and "log".
# In particular, a bin edge of 0 doesn't play well with "log".
f1.set_pdf_params(bin_type=['log', 'lin'],
    bin_range=[[5e4, 5.5e13], [1., 10000.]],
    bin_number=[1000, 1000])

# Runs the functions.
tpf.run_generator()

# This calculates the M in RMS and writes out a text file with
# the RMS values and the lengths. The R happens because sqrt=[True, False]
# in add_function.
# The file is named 'rms_vel_D.txt'. It will sqrt only the MS velocity column.
tpf.write_out_means()
# Writes out the raw PDF bins and bin edges to a HDF5 file.
# The file is named 'rms_vel_D.h5'.
tpf.write_out_arrays()
```

3.5.8 Two-Point Correlation Functions

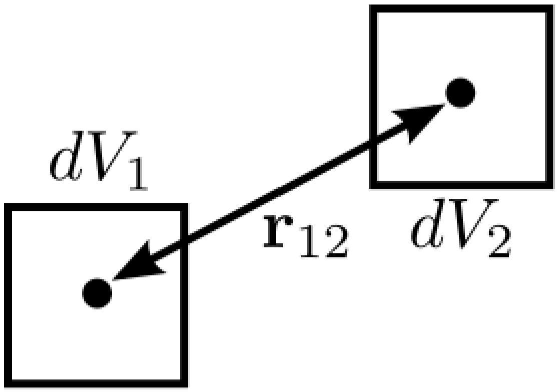
In a Gaussian random field of galaxies, the probability of finding a pair of galaxies within the volumes dV_1 and dV_2 is

$$dP = n^2 dV_1 dV_2$$

where n is the average number density of galaxies. Real galaxies are not distributed randomly, rather they tend to be clustered on a characteristic length scale. Therefore, the probability of two galaxies being paired is a function of radius

$$dP = n^2(1 + \xi(\mathbf{r}_{12}))dV_1 dV_2$$

where $\xi(\mathbf{r}_{12})$ gives the excess probability as a function of $\mathbf{r}_{12}$, and is the two-point correlation function. Values of ξ greater than one mean galaxies are super-gaussian, and visa-versa. In order to use the TPF to calculate two point correlation functions, the number of pairs of galaxies between the two dV volumes is measured. A PDF is built that gives the probabilities of finding the number of pairs. To find the excess probability, a function *write_out_correlation* does something similar to *write_out_means* (above), but also normalizes by the number density of galaxies and the dV volumes. As an aside, a good rule of thumb is that for galaxies, $\xi(r) = (r_0/r)^{1.8}$ where $r_0 = 5$ Mpc/h.



It is possible to calculate the correlation function for galaxies using the TPF using a script based on the example below. Unlike the figure above, the volumes are spherical. This script can be run in parallel.

```
from yt.mods import *
from yt.utilities.kdtree import *
from yt.analysis_modules.two_point_functions.api import *

# Specify the dataset on which we want to base our work.
ds = load('data0005')

# Read in the halo centers of masses.
CoM = []
data = file('HopAnalysis.out', 'r')
for line in data:
    if '#' in line: continue
    line = line.split()
    xp = float(line[7])
    yp = float(line[8])
    zp = float(line[9])
    CoM.append(np.array([xp, yp, zp]))
data.close()

# This is the same dV as in the formulation of the two-point correlation.
dV = 0.05
radius = (3./4. * dV / np.pi)**(2./3.)

# Instantiate our TPF object.
# For technical reasons (hopefully to be fixed someday) `vol_ratio`
# needs to be equal to the number of tasks used if this is run
# in parallel. A value of -1 automatically does this.
tpf = TwoPointFunctions(ds, ['x'],
    total_values=1e7, comm_size=10000,
    length_number=11, length_range=[2*radius, .5],
    length_type="lin", vol_ratio=-1)

# Build the kD tree of halos. This will be built on all
# tasks so it shouldn't be too large.
```

(continues on next page)

(continued from previous page)

```
# All of these need to be set even if they're not used.
# Convert the data to fortran major/minor ordering
add_tree(1)
fKD.t1.pos = np.array(CoM).T
fKD.t1.nfound_many = np.empty(tpf.comm_size, dtype='int64')
fKD.t1.radius = radius
# These must be set because the function find_many_r_nearest
# does more than how we are using it, and it needs these.
fKD.t1.radius_n = 1
fKD.t1.nn_dist = np.empty((fKD.t1.radius_n, tpf.comm_size), dtype='float64')
fKD.t1.nn_tags = np.empty((fKD.t1.radius_n, tpf.comm_size), dtype='int64')
# Makes the kD tree.
create_tree(1)

# Remembering that two of the arguments for a function are the raw
# coordinates, we define a two-point correlation function as follows.
def tpcorr(a, b, r1, r2, vec):
    # First, we will find out how many halos are within fKD.t1.radius of our
    # first set of points, r1, which will be stored in fKD.t1.nfound_many.
    fKD.t1.qv_many = r1.T
    find_many_r_nearest(1)
    nfirst = fKD.t1.nfound_many.copy()
    # Second.
    fKD.t1.qv_many = r2.T
    find_many_r_nearest(1)
    nsecond = fKD.t1.nfound_many.copy()
    # Now we simply multiply these two arrays together. The rest comes later.
    nn = nfirst * nsecond
    return nn

# Now we add the function to the TPF.
# ``corr_norm`` is used to normalize the correlation function.
tpf.add_function(function=tpcorr, out_labels=['tpcorr'], sqrt=[False],
    corr_norm=dV**2 * len(CoM)**2)

# And define how we want to bin things.
# It has to be linear bin_type because we want 0 to be in the range.
# The big end of bin_range should correspond to the square of the maximum
# number of halos expected inside dV in the volume.
tpf[0].set_pdf_params(bin_type='lin', bin_range=[0, 2500000], bin_number=1000)

# Runs the functions.
tpf.run_generator()

# Write out the data to "tpcorr_correlation.txt"
# The file has two columns, the first is radius, and the second is
# the value of \xi.
tpf.write_out_correlation()

# Empty the kdtree
del fKD.t1.pos, fKD.t1.nfound_many, fKD.t1.nn_dist, fKD.t1.nn_tags
free_tree(1)
```

If one wishes to operate on field values, rather than discrete objects like halos, the situation is a bit simpler, but still a bit confusing. In the example below, we find the two-point correlation of cells above a particular density threshold. Instead of constant-size spherical dVs, the dVs here are the sizes of the grid cells at each end of the rulers. Because there can be cells of different volumes when using AMR, the number of pairs counted is actually the number of most-refined-cells contained within the volume of the cell. For one level of refinement, this means that a root-grid cell has the equivalent of 8 refined grid cells in it. Therefore, when the number of pairs are counted, it has to be normalized by the volume of the cells.

```
from yt.mods import *
from yt.utilities.kdtree import *
from yt.analysis_modules.two_point_functions.api import *

# Specify the dataset on which we want to base our work.
ds = load('data0005')

# We work in simulation's units, these are for conversion.
vol_conv = ds['cm'] ** 3
sm = ds.index.get_smallest_dx()**3

# Our density limit, in gm/cm**3
dens = 2e-31

# We need to find out how many cells (equivalent to the most refined level)
# are denser than our limit overall.
def _NumDens(data):
    select = data["density"] >= dens
    cv = data["cell_volume"][select] / vol_conv / sm
    return (cv.sum(),)
def _combNumDens(data, d):
    return d.sum()
add_quantity("TotalNumDens", function=_NumDens,
    combine_function=_combNumDens, n_ret=1)
all = ds.all_data()
n = all.quantities["TotalNumDens"]()

print(n,'n')

# Instantiate our TPF object.
tpf = TwoPointFunctions(ds, ['density', 'cell_volume'],
    total_values=1e5, comm_size=10000,
    length_number=11, length_range=[-1, .5],
    length_type="lin", vol_ratio=1)

# Define the density threshold two point correlation function.
def dens_tpcorr(a, b, r1, r2, vec):
    # We want to find out which pairs of Densities from a and b are both
    # dense enough. The first column is density.
    abig = (a[:,0] >= dens)
    bbig = (b[:,0] >= dens)
    both = np.bitwise_and(abig, bbig)
    # We normalize by the volume of the most refined cells.
    both = both.astype('float')
    both *= a[:,1] * b[:,1] / vol_conv**2 / sm**2
    return both

# Now we add the function to the TPF.
# ``corr_norm`` is used to normalize the correlation function.
tpf.add_function(function=dens_tpcorr, out_labels=['tpcorr'], sqrt=[False],
    corr_norm=n**2 * sm**2)

# And define how we want to bin things.
# It has to be linear bin_type because we want 0 to be in the range.
```

(continues on next page)

(continued from previous page)

```
# The top end of bin_range should be 2^(2l)+1, where l is the number of
# levels, and bin_number=2^(2l)+2
tpf[0].set_pdf_params(bin_type='lin', bin_range=[0, 2], bin_number=3)

# Runs the functions.
tpf.run_generator()

# Write out the data to "dens_tpcorr_correlation.txt"
# The file has two columns, the first is radius, and the second is
# the value of \xi.
tpf.write_out_correlation()
```