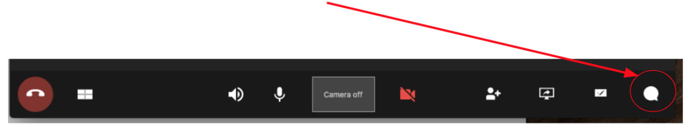
Working with data objects in YT

ALEX RAZOUMOV
alex.razoumov@westgrid.ca

To ask questions

- Websteam: email **info@westgrid.ca**
- Vidyo: use the GROUP CHAT to ask questions



- Please mute your mic unless you have a question
- Feel free to ask questions via audio at any time

Let's review: HTTP://YT-PROJECT.ORG

- Python package for analyzing and visualizing volumetric, multi-resolution data
  - really a library for non-interactive use, does not offer 3D interactivity found in such tools as ParaView and VisIt
    - there is an ongoing project VIEWYT to develop Qt widgets for interacting with YT plots
  - discretization: structured, unstructured, variable-resolution (curvilinear), particle data
  - very easy to learn, wonderful documentation at `https://yt-project.org/doc`
  - great for batch off-screen rendering (including HPC clusters); parallelized with `mpi4py`
- Initially written for analysing *Enzo* output data, adapted to understand other data formats from astrophysics and beyond
  - documentation strongly focused on astrophysical data (do not let this deter you)
  - currently has readers for $\sim$ 25 file formats
  - can import generic data on uniform and AMR (nested) grids, particles, unstructured meshes
  - areas: astrophysics, seismology, nuclear engineering, molecular dynamics, oceanography
- Strong data-processing capabilities (today's focus)

Covered in Part 1 (Nov-21)

Slides and recording at
`https://westgrid.github.io/trainingMaterials/tools/visualization`

- Historical context and overview of supported data formats
  - can read output of many astrophysical codes
  - with data already in Python, can create YT-native datasets containing uniform grids, AMR grids, semi-structured (hexahedral) grids, unstructured grids, particle data
- Installing YT with conda, pip, from source
- Loading and examining data: domain parameters, fields, AMR subgrids
- Slice plots
- Projection plots
- Volume rendering
  - creating scenes
  - transfer functions: manual/automatic Gaussians, custom continuous colourmaps, using defaults
  - controlling the scene camera: zooming in, moving focus, rotating
- Installing YT in user space on CC clusters + parallel rendering with `mpi4py`
- Working with generic uniform array data and generic AMR data
- Time-series analysis (working with time-dependent data)

Review: rotating a cosmological volume with grid annotations

More on parallel YT at `https://yt-project.org/doc/analyzing/parallel_computation.html`

1. Download/uncompress the data from `http://yt-project.org/data`
2. On the cluster, save this as `grids.py`:

```
import yt, numpy as np
yt.enable_parallelism()    # turn on MPI parallelism via mpi4py
ds = yt.load("Enzo_64/DD0043/data0043")
sc = yt.create_scene(ds, ('gas', 'density'))
cam = sc.camera
cam.resolution = (1024, 1024)    # resolution of each frame
sc.annotate_domain(ds, color=[1, 1, 1, 0.005])    # draw the domain boundary [r,g,b,alpha]
sc.annotate_grids(ds, alpha=0.005)    # draw the grid boundaries
sc.save('frame0000.png', sigma_clip=4)
for i in cam.iter_rotate(np.pi, 900):    # rotate by 180 degrees over 900 frames
    sc.save('frame%04d.png' % (i+1), sigma_clip=4)
```

3. Write the job submission script `yt-mpi.sh`:

```
#!/bin/bash
#SBATCH --time=12:00:00    # walltime in d-hh:mm or hh:mm:ss format
#SBATCH --ntasks=4         # number of MPI processes
#SBATCH --mem-per-cpu=3800
#SBATCH --account=...
source /home/razoumov/astro/bin/activate
srun python nested.py
```

Review: rotating a cosmological volume (cont.)

- Submit the job

```
$ sbatch yt-mpi.sh
```

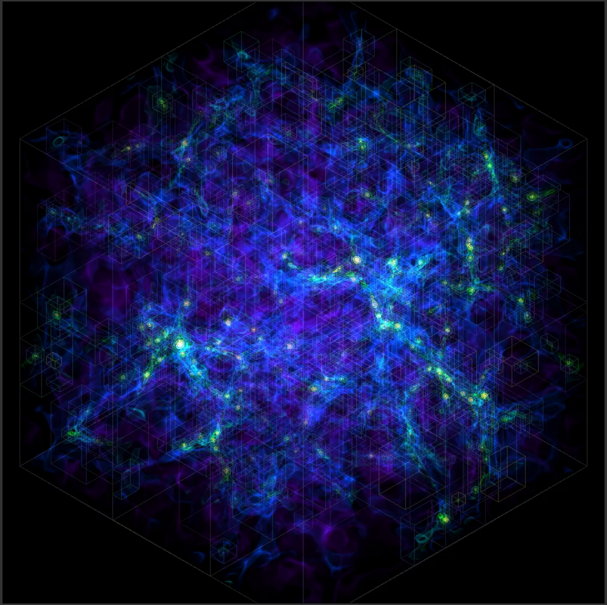
- Performance: serial ➢ 1.47 frames/min., parallel on 4 cores ➢ 4.05 frames/min.
- Make a Quicktime-compatible MP4 right on the cluster

```
$ ffmpeg -r 30 -i frame%04d.png -c:v libx264 -pix_fmt yuv420p -vf \
      "scale=trunc(iw/2)*2:trunc(ih/2)*2" grids.mp4
```

- Download it to your laptop

```
$ rsync -av --progress cedar.computecanada.ca:path/to/grids.mp4 .
```

Review: rotating a cosmological volume (cont.)



- Online (highly compressed) `https://vimeo.com/301503962`
- On presenter's laptop `grids.mp4`

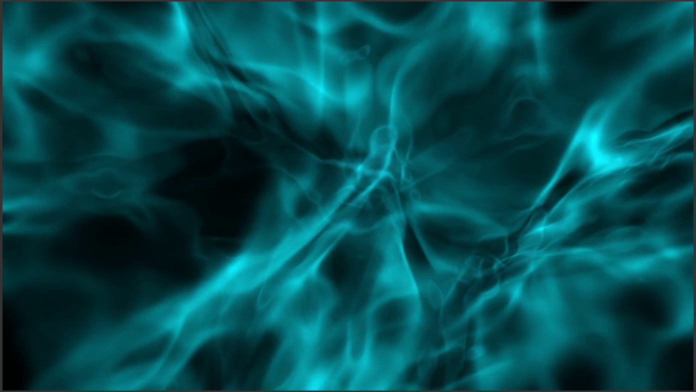
Another volume rendering animation: deeply nested zoom

1. Download/uncompress the data from `http://yt-project.org/data`
2. On the cluster, save this as `nested.py`:

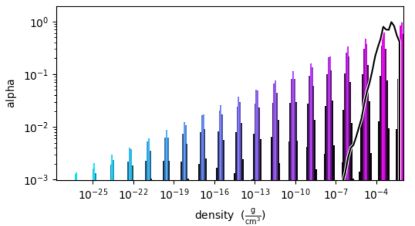
```
import yt, numpy as np; yt.enable_parallelism()
ds = yt.load('DeeplyNestedZoom/DD0025/data0025')
initialWidth = float(ds.domain_width.in_units('kpc')[0])    # 97.8 kpc on a side
rho, c = ds.find_max("density")    # find the highest density peak (value and location)

sc = yt.create_scene(ds); sc.camera.resolution = (1920, 1080)
sc.camera.set_focus(c)                # focus on the highest density peak
source, bounds = sc[0], (2e-28, 1e-2)            # very large range of densities
source.set_field('density')          # field to render
tf = yt.ColorTransferFunction(x_bounds=np.log10(bounds))
tf.add_layers(N=20, w=0.03, colormap='cool')     # add 20 Gaussians filters
source.tfh.tf, source.tfh.bounds = tf, bounds    # tfh stands for TransferFunctionHelper
source.tfh.plot('transferFunction.png', profile_field='density')

# in 1795 log steps change the window from 97.8 kpc down to 9.78e-11 kpc = 0.0202 AU = 3.02e6 km
for i, coef in enumerate(np.linspace(start=0,stop=12,num=1795)):    # i=0..1794
    width = initialWidth/10.**coef    # width of the visualization window
    sc.camera.set_width(((width*192/108,'kpc'),(width,'kpc'),(width,'kpc')))
    sc.save('frame%04d.png' % (i+1), sigma_clip=4)
```

3. Modify the job submission script accordingly and submit it to the queue

Deeply nested zoom (cont.)



More on volume rendering at `http://bit.ly/2HkPS3L`



- Pick your favourite colourmap `yt.show_colormaps()`
- In 1795 logarithmic steps changing the window from 97.8 kpc (diameter of a large spiral galaxy) down to 9.78e-11 kpc = 0.0202 AU = 3.02e6 km $\approx 2\,R_\odot$
- Encoded at 60fps
- Online (highly compressed) `https://vimeo.com/312290924`
- On presenter's laptop `nested.mp4`

Today: using YT for data analysis and processing

More on YT's data objects `https://yt-project.org/doc/analyzing/objects.html`

- Subsetting data in many different ways
- Creating iso- and other surfaces, exporting them as 3D scenes to interactive viewers
- Creating streamlines
- We'll view results with Matplotlib, plot.ly, ParaView, Sketchfab

In YT you can easily create new objects from existing data. These new objects can define subsets, derivative datasets, collections with certain properties.

Creating a flat collection of all points

```
>>> import yt
>>> ds = yt.load("IsolatedGalaxy/galaxy0030/galaxy0030")

>>> ds.domain_width    # [1., 1., 1.] code_length
YTArray([1., 1., 1.]) code_length
>>> ds.domain_width.in_units('Mpc')
YTArray([1.00010449, 1.00010449, 1.00010449]) Mpc

>>> ds.index.num_grids
173
>>> ds.index.grid_levels
array([[0], [1], [1], ..., [8], [8]], dtype=int32)
>>> ds.index.grid_dimensions
array([[32,32,32], [16,18,16], [16,18,16], ..., [8,16,20], [8,12,12]], dtype=int32)

>>> all = ds.all_data()    # create a flat collection of all points
>>> all.fcoords    # array of xyz coordinates of all points
YTArray([[0.015625, 0.015625, 0.015625],
         [0.015625, 0.015625, 0.046875],
         [0.015625, 0.015625, 0.078125],
         ...,
         [0.498962, 0.49749756, 0.49981689],
         [0.498962, 0.49749756, 0.49993896]]) code_length
```

Examining this flat collection

```
>>> all.index.num_grids
173
>>> all.index.grid_levels   # array of refinement levels (one per grid)
array([[0], [1], [1], ..., [8], [8]], dtype=int32)
>>> all.min_level, all.max_level   # lowest and highest refinement levels
(0, 8)
>>> all.index.grid_dimensions   # list dimensions of all subgrids
array([[32,32,32], [16,18,16], [16,18,16], ..., [8,16,20], [8,12,12]], dtype=int32)

>>> all.size   # does not get filled until you call all.fcoords or similar
3,644,460
>>> all.ires   # array of refinement levels in the flat collection (one per cell)
array([0, 0, 0, ..., 8, 8, 8])
>>> all.index.field_list   # list all variables
[..., ('enzo', 'Density'), ..., ('enzo', 'Temperature'), ...]

>>> all['density']   # 1D array of densities
YTArray([4.92775113e-31, 4.94005233e-31, ..., 1.59561490e-25, 1.09824903e-24]) g/cm**3
>>> all.min('density'), all.max('density')
(8.472937507539987e-32 g/cm**3, 7.73426503924e-24 g/cm**3)
>>> all.quantities.max_location('density')   # the highest density and its location in cm
>>> all.quantities.center_of_mass()
>>> all.quantities.angular_momentum_vector()
>>> all.quantities.total_mass().in_units('Msun')
```

'.r' indexer: a handy tool to reference the entire region or its subset

YT provides a special indexer that lets you use certain indexing schemes

- this is not a function, but rather an attribute that exposes a particular slicing interface
- will typically output a flat array, or a 3D array if specifying fixed resolution
- in a way, similar to Pandas's indexers '.loc' and '.iloc'

```
>>> volume = ds.r[:,:,:]
>>> volume.shape
(3644460,)
>>> volume['density']
YTArray([4.92775113e-31, 4.94005233e-31, 4.93824694e-31, ...,
         1.12879234e-25, 1.59561490e-25, 1.09824903e-24]) g/cm**3

>>> rho = ds.r['density']    # stored as a flattened 1D array with all data
>>> type(rho)
<class 'yt.units.yt_array.YTArray'>
>>> rho.shape    # 3,644,460 cells
(3644460,)
>>> type(rho.d)
<class 'numpy.ndarray'>

>>> slab = ds.r[(100, 'kpc'):(200,'kpc'),:,:]    # flattened 1D array of all points with 100kpc < x < 200kpc
>>> len(slab.fcoords)    # 3,072 points (denser regions not included)
3072
```

'.r' indexer (cont.)

```
>>> region = ds.r[::21j, ::35j, ::100j]  # return the entire domain as a fixed-resolution grid
>>> region.shape
(21, 35, 100)
>>> region['density'].shape
(21, 35, 100)
>>> region['density'][10,5,30]
4.970245638988166e-31 g/cm**3

>>> slc = ds.r[:,:,0.25]     # slice stored as a flattened 1D array
>>> slc.fcoords
YTArray([[0.015625,0.015625,0.265625], ..., [0.4921875,0.4921875,0.2578125]]) code_length
>>> slc.shape    # might not be filled until you call slc.fcoords
(1864,)

>>> frb = slc.to_frb(width=ds.domain_width[0], resolution=1024)    # project onto a 2D fixed-res buffer
>>> frb.limits
{'x': (0.0 code_length, 1.0 code_length), 'y': (0.0 code_length, 1.0 code_length), 'z': None}
>>> frb['density'].shape
(1024, 1024)

>>> from matplotlib import pyplot as plt; from numpy import log10
>>> plt.imshow(log10(frb['density'].d))    # convert to numpy array and calculate log10
>>> plt.colorbar()
>>> plt.savefig('slice.png', dpi=200)
```

Spherical regions

```
>>> sphere = ds.sphere(center='max', radius=(0.1,'Mpc'))   # include points within 0.1Mpc of max density
>>> sphere.fcoords   # x,y,z of these points
YTArray([[0.50012207,0.49621582,0.5098877], ..., [0.4989624,0.49749756,0.49993896]]) code_length

>>> sphere.size    # this attribute and next might not be available until you start accessing the data
2582726
>>> sphere.shape   # really a 1D flattened array
(2582726,)

>>> sphere.max('density')
7.73426503924e-24 g/cm**3
>>> sphere['density']
YTArray([2.05686132e-27, 1.98130330e-27, ..., 1.59561490e-25, 1.09824903e-24]) g/cm**3

>>> tiny = ds.sphere(center='max', radius=(0.2, 'kpc'))   # define a much smaller sphere
>>> tiny.fcoords.shape   # only 19 points
(19, 3)
>>> for i in range(tiny['temperature'].size):
...     print('(%.5e,  %.5e,  %.5e)    %f' %
...           (tiny['x'][i], tiny['y'][i], tiny['z'][i], tiny['temperature'][i]))
...
(1.55524e+24,  1.54206e+24,  1.54357e+24)    13975.521484
...
(1.55600e+24,  1.54281e+24,  1.54357e+24)    11659.029297
```

Spherical regions (cont.)

```
>>> smallerSphere = ds.sphere(center="max", radius=(0.09, "Mpc"))    # 2,436,755 points
>>> smallerSphere.fcoords.shape
(2436755, 3)

>>> sphere.size
2582726

>>> shell = sphere - smallerSphere     # only points in the shell 0.09Mpc-0.1Mpc
>>> shell.fcoords.shape
(145971, 3)   # 145,971 points
```

Data selection based on the value of one or more fields

```
>>> volume = ds.r[:,:,:]
>>> volume.fcoords.shape   # the original 3,644,460 points
(3644460, 3)
>>> volume.quantities.extrema('density')
YTArray([8.47293751e-32, 7.73426504e-24]) g/cm**3

# only include points denser than 1e-24 g/cm**3
>>> dense = volume.cut_region(field_cuts=["obj['density'] > 1e-24"])
>>> dense.fcoords.shape   # 11,747 such points
(11747, 3)

>>> dense['temperature']
YTArray([ 9730.06542969,  6468.8828125,  9101.88769531, ...,
         10117.41601562,  9845.79492188, 10173.02148438]) K

>>> denseAndHot = volume.cut_region(field_cuts=["obj['density'] > 1e-24",
                                                "obj['temperature'] > 1e5"])
>>> denseAndHot.fcoords.shape
(4, 3)
>>> denseAndHot['temperature']
YTArray([113957.3125, 104528.1796875, 104617.59375, 100997.421875]) K
```

Creating iso-surfaces

Let's create a density isosurface

```
>>> surface = ds.surface(data_source=volume, surface_field="density", field_value=1e-27)
>>> surface['density'].size        # 168,235 triangle-centered values
168235
>>> surface['density'].min(), surface['density'].max()    # the surface is approximate
(9.549593893136611e-28 g/cm**3, 1.0326803047107872e-27 g/cm**3)

>>> surface['temperature'].min(), surface['temperature'].max() # other fields available too
(11850.747686367733 K, 13641.066390620443 K)

>>> surface.vertices.shape   # three 1D arrays (x,y,z) of 504,705 vertices
(3, 504705)
>>> surface.vertices
YTArray([[0.5, 0.50018984, ..., 0.49951172, 0.49995156],
         [0.501835, 0.50195312, ..., 0.5, 0.5],
         [0.5234375, 0.5234375, ..., 0.48079189, 0.48046875]]) code_length

>>> surface.triangles.shape   # 168,235 triangles * 3 vertices * (x,y,z) each
(168235, 3, 3)
```

Exporting polygonal surfaces

And now let’s export our surface

```
>>> surface.export_ply(filename='surface.ply', color_field='temperature')    # older PLY (Polygon File Format)

>>> mi, ma = min(surface['temperature']), max(surface['temperature'])
>>> print(mi,ma)
11850.747686367733 K 13641.066390620443 K
>>> surface.export_obj(filename='surface', transparency=1.0, color_field_min = mi, color_field_max = ma,
                       color_field='temperature')    # will create an OBJ file and an MTL file

>>> surface.export_sketchfab(title='test', description='quick test', color_field='temperature')
Model uploaded to: https://sketchfab.com/models/03770412dd1547f3b14a4f7b7c5afbf7
```

- While all of these let you store a field at every polygonal face, in all these methods the sampled field is stored via its *very approximate colour* $\Rightarrow$ in the visualization you can’t access the field value explicitly $\Rightarrow$ more for outreach and communication, than scientific visualization
  - in OBJ+MTL file colours are stored as distinct materials `http://bit.ly/2W8tzlq` (Wikipedia)
- As far as I can tell, `surface.export_obj()` and `surface.export_blender()` produce identical results
- `surface.export_sketchfab()` exports surfaces to 3D hosting platform `https://sketchfab.com`
  - to be explored with a WebGL viewer in a browser
  - view our interactive Sketchfab model at `http://bit.ly/2Wbp49M`

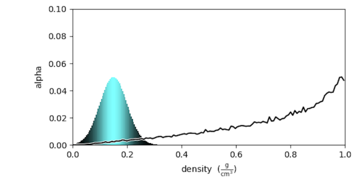
Creating a surface at a fixed geometric location

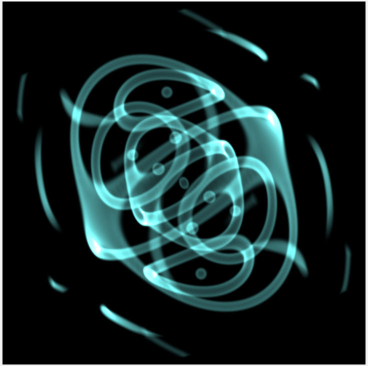
In astrophysical YT datasets there is a built-in `radius` field
= distance from the center of the computational volume

```
>>> import yt
>>> ds = yt.load("IsolatedGalaxy/galaxy0030/galaxy0030")

>>> all = ds.all_data()     # flat collection of all points
>>> all.index.field_list    # shows 55 fields

>>> all['radius']
YTArray([2.58903707e+24, 2.53458226e+24, 2.48268038e+24, ...,
         8.41293304e+21, 8.37912906e+21, 8.36217582e+21]) cm
>>> all['radius'].shape
(3644460,)

>>> surface = ds.surface(data_source=all, surface_field='radius', field_value=0.5)    # in code units

>>> ds.domain_width.in_units('kpc')
YTArray([1000.10448889, 1000.10448889, 1000.10448889]) kpc
>>> surface = ds.surface(data_source=all, surface_field='radius', field_value=(300,'kpc'))
```

More on astrophysical fields
http://yt-project.org/doc/analyzing/fields.html

Creating a surface at a fixed geometric location (cont.)

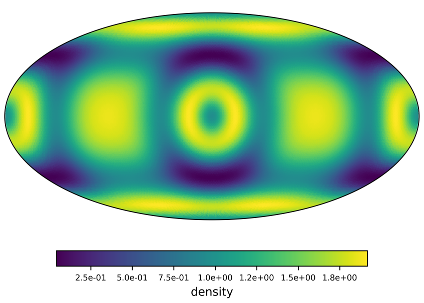




Recall `sineEnvelope.nc` dataset (Part 1)

```
import yt, numpy as np, healpy as hp
from netCDF4 import Dataset
import matplotlib.pyplot as plt, matplotlib.tri as tri
vol = Dataset('sineEnvelope.nc', 'r')
rho = vol.variables['density'][:,:,:] # 100^3 numpy array

# create coordinate arrays x,y,z (each a 3D array)
[x,y,z] = np.mgrid[5e-3:1:0.01,5e-3:1:0.01,5e-3:1:0.01]
x.shape                 # (100,100,100) array
x.min(), x.max()        # (0.005, 0.995) in each dimension

# compute distance from the centre
r = np.sqrt((x-0.5)**2+(y-0.5)**2+(z-0.5)**2)

# create a yt-native dataset
data = dict(density = rho, radius=r)
bbox = np.array([[0,1],[0,1],[0,1]])
ds = yt.load_uniform_grid(data=data, bbox=bbox,
        domain_dimensions=rho.shape, length_unit=1.)

ds.index.field_list
# shows [('stream', 'density'), ('stream', 'radius')]
```

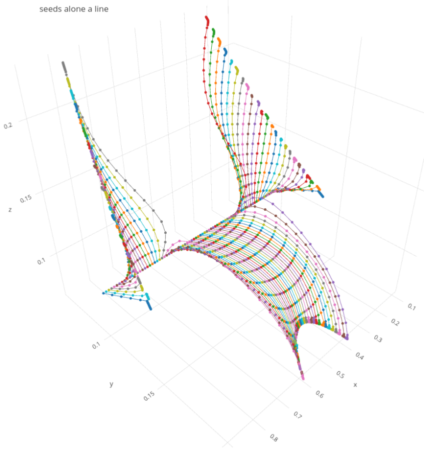
Creating a surface at a fixed geometric location (cont.)



On presenter's laptop `sphere.png`

```
all = ds.all_data()    # flat collection of all points
surf = ds.surface(data_source=all,
   surface_field=('stream', 'radius'), field_value=0.3)
numTriangles = surf.triangles.shape[0]
print(numTriangles, 'surface triangles')    # 33,704 triangles

coords = np.zeros((numTriangles,3))
for i in range(3):
    coords[:,i] = (surf.triangles[:,0,i] +
       surf.triangles[:,1,i] + surf.triangles[:,2,i])/3.

# compute (lat,long) of all trianles as seen from the centre
coords -= 0.5    # move the centre of the box to (0,0,0)
theta, phi = hp.vec2ang(coords)
theta -= np.pi/2.    # for plotting theta should be [-pi/2,pi/2]
phi -= np.pi         #                phi should be [-pi,pi]
den = np.array(surf['density'])

# a pseudocolor plot of the unstructured triangular mesh
plt.clf(); plt.cla(); plt.close()    # reset the plot
triang = tri.Triangulation(phi,theta)
ax = plt.subplot(111, projection = 'mollweide')
im = ax.tripcolor(triang,den)
frame = plt.gca(projection='mollweide')
frame.axes.xaxis.set_ticklabels([])
frame.axes.yaxis.set_ticklabels([])
cbar = plt.colorbar(im, orientation="horizontal", pad=0.1,
                    shrink=0.75, format='%.1e')
cbar.set_label('density')
cbar.ax.tick_params(labelsize=7)
plt.savefig('sphere.png', dpi=600)
```

Streamlines




```
grad_fields = ds.add_gradient_fields(('stream', 'density'))
grad_fields     # new fields 'density_gradient_{x,y,z}'

from yt.visualization.api import Streamlines
# import plotly.offline as py    # offline plotting
import plotly.plotly as py       # online plotting
import plotly.graph_objs as go

seeds = np.zeros((81,3))    # x,y,z for 81 points along a line
seeds[:,0]  = np.linspace(0.1,0.9,81)
seeds[:,1], seeds[:,2] = 0.1, 0.1

streamlines = Streamlines(ds, seeds, 'density_gradient_x',
        'density_gradient_y', 'density_gradient_z')
streamlines.integrate_through_volume()

data = []
for stream in streamlines.streamlines:
    spheres = go.Scatter3d(x=stream[:,0], y=stream[:,1],
                           z=stream[:,2], marker=dict(size=3))
    data.append(spheres)

layout = go.Layout(height=1200, width=1200,
              title='seeds alone a line', showlegend=False)
fig = go.Figure(data=data, layout=layout)
py.plot(fig)
```

- The data selector along a streamline is not yet implemented
- Check the 3D interactive plot `http://bit.ly/2R8dTdZ`

Summary

- Plotting: slices, projections, volume rendering (covered in Part 1)
- On the cluster can script your entire visualization as a batch off-screen CPU rendering job with `mpi4py` parallelization
- Today we saw a number of interactions with data objects
- This is Python, so the sky is the limit!
- Excellent documentation at `https://yt-project.org/doc`

Questions?