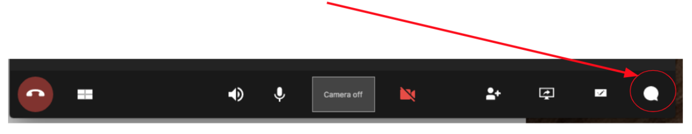
Using YT for analysis and visualization of volumetric data

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

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
  - 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)
  - areas: astrophysics, seismology, nuclear engineering, molecular dynamics, oceanography
- This presentation is really targeted at non-astrophysicists
  - astro folks are already aware of YT and either use it, or have a good reason not to
  - researchers from many other fields deal with 3D volumetric data, could benefit from YT

Historical context

- Astrophysicists have been running multi-dimensional simulations for the last 40+ years, needed a tool to visualize data
- In the early 2000s no access to good, open-source, multi-platform, scalable multi-dimensional scientific visualization tools
  - many researchers developed visualization and data analysis workflows in IDL and MATLAB (both commercial with lots of limitations)
  - ParaView and VisIt were only starting, not yet known and/or regarded as too general-purpose or too large by many
- Many wrote tools (IFrIT, FTTE, ...) for their own use, some were open-sourced
- One of these tools was `YT`, initially written by Matthew Turk around ∼2007
  - Python-based ⇒ high-level abstractions for data manipulation!
  - intuitive interface
  - soon invited other researchers to use and develop it
  - currently 128 listed contributors

Data formats

ART	ARTIO	Athena	Athena++	AMReX
Pluto	Enzo	Enzo-P	Exodus II	FITS
FLASH	Gadget	GAMER	Generic AMR	Generic array
Semi-structured (hexahedral) grid	Unstructured grid	Generic particle	Gizmo	Halo catalog
openPMD	PyNE	RAMSES	SPH Particle	Tipsy

Installing YT and add-ons (data, tutorial Jupyter Notebooks)

install YT itself

```
$ conda install -c conda-forge yt
or `pip install yt`
or `git clone https://github.com/yt-project/yt && cd yt && python setup.py develop`
```

```
$ cd /tmp/yt-data
$ url=http://yt-project.org/data
$ wget $url/enzo_tiny_cosmology.tar.gz   # 32^3 base + 5 additional AMR levels
$ wget $url/Enzo_64.tar.gz               # 64^3 base + 4 additional AMR levels
$ wget $url/IsolatedGalaxy.tar.gz        # 32^3 base + 8 additional AMR levels
$ wget $url/MOOSE_Sample_data.tar.gz  # unstructured mesh (from a finite-element code)
$ wget $url/ArepoBullet.tar.gz
$ wget $url/geos.tar.gz
$ wget $url/DICEGalaxyDisk_nonCosmological.tar.gz
unpack them
```

```
$ git clone https://github.com/yt-project/yt
$ cd yt/doc/source/quickstart
$ yt notebook
```

Loading and examining data

```
>>> import yt
>>> ds = yt.load('/tmp/yt-data/IsolatedGalaxy/galaxy0030/galaxy0030')
yt : [INFO ] 2018-11-17 17:36:02,879 Parameters: current_time       = 0.0060000200028298
yt : [INFO ] 2018-11-17 17:36:02,879 Parameters: domain_dimensions         = [32 32 32]
yt : [INFO ] 2018-11-17 17:36:02,879 Parameters: domain_left_edge          = [0. 0. 0.]
yt : [INFO ] 2018-11-17 17:36:02,879 Parameters: domain_right_edge         = [1. 1. 1.]
yt : [INFO ] 2018-11-17 17:36:02,880 Parameters: cosmological_simulation   = 0.0

>>> ds.print_stats()       # actually read the mesh, print stats
Parsing Hierarchy : 100% 173/173 [00:00<00:00, 9581.22it/s]
yt : [INFO ] 2018-11-17 17:36:02,908 Gathering a field list (this may take a moment.)
level   # grids        # cells        # cells^3
----------------------------------------------
  0          1          32768               32
  1          8          34304               33
  2          8         181888               57
  3          8         646968               87
  4         15         947856               99
  5         51         874128               96       Smallest Cell:
  6         18         786328               93               Width: 1.221e-04 Mpc
  7         28         446776               77               Width: 1.221e+02 pc
  8         36         209400               60               Width: 2.518e+07 AU
----------------------------------------------               Width: 3.767e+20 cm
           173        4160416

t = 6.00002000e-03 = 1.39768066e+16 s = 4.42898275e+08 years
```

Loading and examining data: dataset fields

```
>>> ds.field_list     # list all 55 dataset fields (grouped into categories)
[('all','creation_time'), ('all','dynamical_time'), ('all','metallicity_fraction'),
('all','particle_index'), ('all','particle_mass'), ('all','particle_position_x'),
('all','particle_position_y'), ('all','particle_position_z'), ('all','particle_type'),
('all','particle_velocity_x'), ('all','particle_velocity_y'), ('all','particle_velocity_z'),
('enzo','Average_creation_time'), ('enzo','Bx'), ('enzo','By'), ('enzo','Bz'),
('enzo','Cooling_Time'), ('enzo','Dark_Matter_Density'), ('enzo','Density'),
('enzo','Electron_Density'), ('enzo', 'Forming_Stellar_Mass_Density'),
('enzo','Galaxy1Colour'), ('enzo','Galaxy2Colour'), ('enzo', 'HII_Density'),
('enzo','HI_Density'), ('enzo','HeIII_Density'), ('enzo','HeII_Density'),
('enzo','HeI_Density'), ('enzo','MBHColour'), ('enzo','Metal_Density'),
('enzo','PhiField'), ('enzo','Phi_pField'), ('enzo','SFR_Density'),
('enzo','Star_Particle_Density'), ('enzo','Temperature'), ('enzo','TotalEnergy'),
('enzo','gammaHI'), ('enzo','kphHI'), ('enzo','kphHeI'), ('enzo','kphHeII'),
('enzo','x-velocity'), ('enzo','y-velocity'), ('enzo','z-velocity'), ('io','creation_time'),
('io','dynamical_time'), ('io','metallicity_fraction'), ('io','particle_index'),
('io','particle_mass'), ('io','particle_position_x'), ('io','particle_position_y'),
('io','particle_position_z'), ('io','particle_type'), ('io','particle_velocity_x'),
('io','particle_velocity_y'), ('io','particle_velocity_z')]

>>> len(ds.derived_field_list)      # list all 405 dataset + derived fields

>>> print(ds.field_info["gas", "vorticity_x"].get_source())    # see derived field definition
```

Loading and examining data: domain parameters

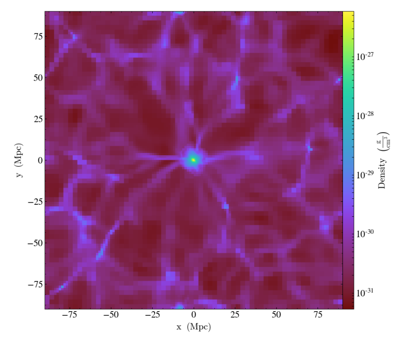
```
>>> ds.domain_width          # box size in code units
YTArray([1., 1., 1.]) code_length

>>> ds.domain_center         # in code units
YTArray([0.5, 0.5, 0.5]) code_length

>>> ds.domain_left_edge      # in code units
YTArray([0., 0., 0.]) code_length

>>> ds.domain_right_edge     # in code units
YTArray([1., 1., 1.]) code_length

>>> ds.domain_dimensions     # base grid size
array([32, 32, 32])

>>> ds.parameters      # list all 402 simulation parameters (outputs a Python dictionary)

>>> ds.parameters['StarMakerMinimumMass']
8000

>>> print(ds.domain_width.in_units("kpc"))
[1000.10448889 1000.10448889 1000.10448889] kpc

>>> print(ds.domain_width.in_units("au"))
[2.06286359e+11 2.06286359e+11 2.06286359e+11] au
```

Loading and examining data: subgrids in the AMR hierarchy

```
>>> print(ds.index.num_grids, ds.index.max_level)
173 8

>>> ds.index.grid_levels
array([[0], [1],
       ...
       [8], [8]], dtype=int32)

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

>>> ds.index.grid_left_edge
YTArray([[0., 0., 0.], [0.25, 0.21875, 0.25],
        ...
        [0.49902344, 0.49560547, 0.49755859],
        [0.49804688, 0.49609375, 0.49853516]])
        code_length

>>> ds.index.grid_right_edge
YTArray([[1., 1., 1.], [0.5, 0.5, 0.5],
        ...
        [0.5 , 0.49755859, 0.5 ],
        [0.49902344, 0.49755859, 0.5 ]])
        code_length

>>> ds.index.grid_particle_count
array([[0], [13],
       ...
       [736], [369]], dtype=int32)
```

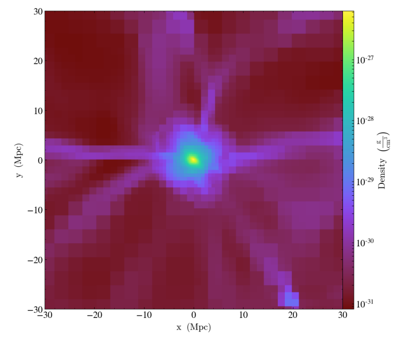
```
>>> g = ds.index.grids[0]
>>> g.ActiveDimensions
array([32, 32, 32], dtype=int32)

>>> g.LeftEdge, g.RightEdge
(YTArray([0., 0., 0.]) code_length,
 YTArray([1., 1., 1.]) code_length)

>>> g.Level
0

>>> g.Children
[EnzoGrid_0002, EnzoGrid_0003, EnzoGrid_0004,
 EnzoGrid_0005, EnzoGrid_0006, EnzoGrid_0007,
 EnzoGrid_0008, EnzoGrid_0009]

# all grids at level=8
>>> gs = ds.index.select_grids(ds.index.max_level)

>>> g2 = gs[0] # select the first of these
>>> print(g2,g2.Parent)
EnzoGrid_0028 EnzoGrid_0023

>>> g2["density"][:,:,0] # subsetting print density
YTArray([[1.0136369e-25, 3.4564638e-25, 6.4192590e-25, ...,
          5.9669651e-25, 5.1001470e-25, 4.2170473e-25],
         ...,
         [9.0090510e-26, 6.8866435e-26, 6.3422531e-26, ...,
          9.1754346e-27, 9.5285530e-27, 1.0011084e-26]],
          dtype=float32) g/cm**3
```

Slice plots



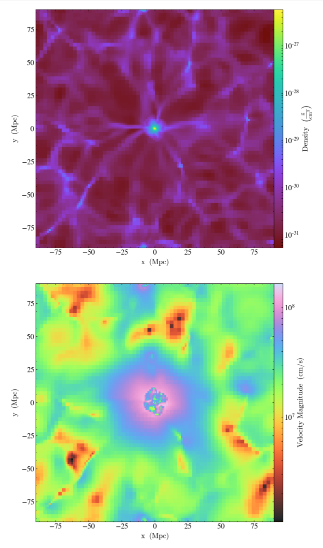

```
import yt

ds = yt.load('/tmp/yt-data/Enzo_64/DD0043/data0043')
print('Redshift =', ds.current_redshift)

slc = yt.SlicePlot(ds, normal='z', fields='density',
                   center='max')
slc.save('slice1.png')

# help(yt.SlicePlot)
  # center='c'                # center of the volume
  # center=[0.1,0.3,0.5]      # specific coordinates
  # center='max'              # highest density point
  # center=('min','temperature')    # lowest temp. point
  # center=('max','dark_matter_density')
```

Slice plots: zoom and window size



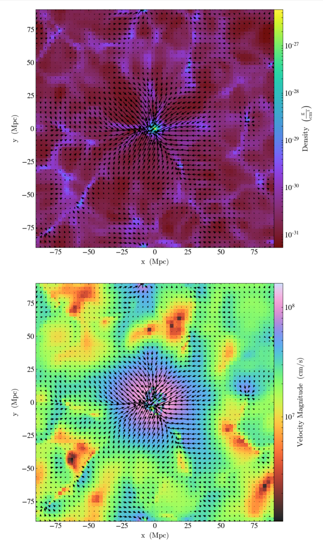

```
slc.zoom(5)
slc.save('slice2.png')

# from yt.units import kpc, Mpc
# slc.set_width(10*Mpc)      # set box size
# slc.save('slice2a.png')

# yt.SlicePlot(ds, normal='z', fields='density', center='max',
#              width=20*Mpc).save('slice2b.png')
```

Slice plots: multiple fields and non-default colourmaps



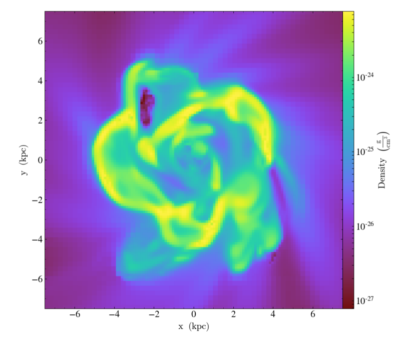

```
slc = yt.SlicePlot(ds, normal='z',
                   fields=['density', 'velocity_magnitude'],
                   center='max')
slc.set_cmap('velocity_magnitude', 'kamae')
slc.save()    # save two images
```

Slice plots: velocity annotations



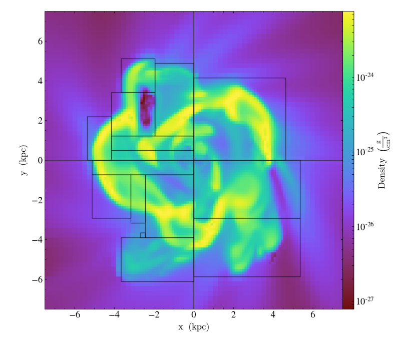

```
slc.annotate_velocity()   # add velocity arrows
slc.save()
```

Slice plots: width as argument



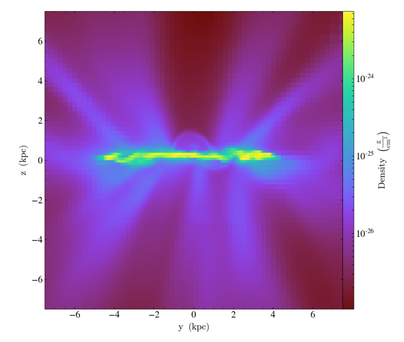

```
ds = yt.load("/tmp/yt-data/IsolatedGalaxy/galaxy0030/galaxy0030")
yt.SlicePlot(ds, 'z', 'density', center='c',
             width=15*kpc).save('slice5.png')    # disk-on
```

Slice plots: grid annotations




```
yt.SlicePlot(ds, 'z', 'density', center='c',
             width=15*kpc).annotate_grids().save('slice5a.png')
```

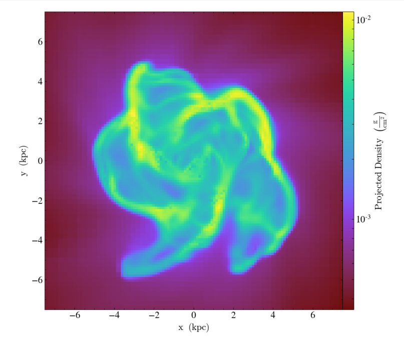
Slice plots: edge-on




```
yt.SlicePlot(ds, 'x', 'density', center='c',
            width=15*kpc).save('slice6.png')    # edge-on
```

Projection: integrate without a weight field

- Density ⇒ column density

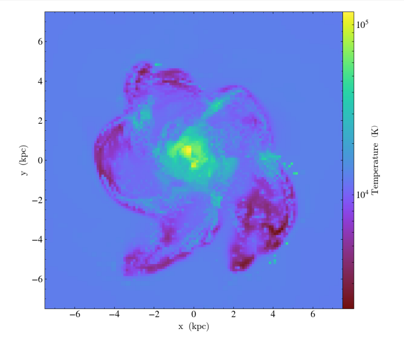



```
import yt
from yt.units import kpc
ds = yt.load('/tmp/yt-data/IsolatedGalaxy/galaxy0030/galaxy0030')

yt.ProjectionPlot(ds, axis='z', fields='density',
                  method='integrate',
                  width=15*kpc).save('projection1.png')

# help(yt.ProjectionPlot)
```

Projection: integrate along the line of sight with a weight field

- Temperature ⇒ mass-weighted temperature

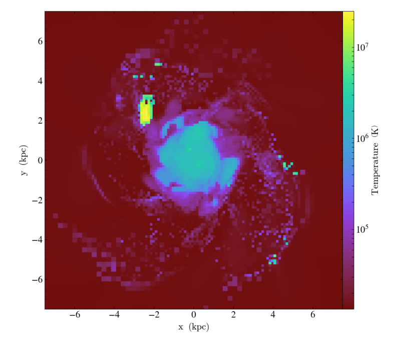
$$\langle T \rangle = \frac{\int_{\mathrm{LOS}} \rho\, T\, dl}{\int_{\mathrm{LOS}} \rho\, dl}$$




```
yt.ProjectionPlot(ds, axis='z', fields='temperature',
                  method='integrate', width=15*kpc,
        weight_field='density').save('projection2.png')
```

Projection: pick out the maximum value along the line of sight

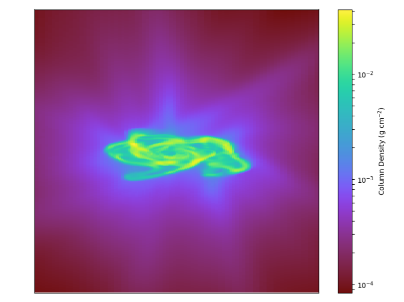
- Obviously, not a physical quantity, but still very useful for identifying unusual regions




```
yt.ProjectionPlot(ds, axis='z', fields='temperature',
                  method='mip',
                  width=15*kpc).save('projection3.png')
```

Projection: off-axis

- Integrates through the volume at an arbitrary angle
- Can do a variety of methods with/without a weight
- Returns an image plane instead of a plot

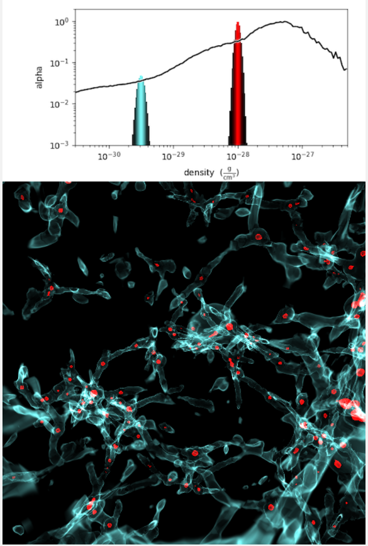



```
image = yt.off_axis_projection(data_source=ds,
           center=[0.5,0.5,0.5], normal_vector=[0,1,0.3],
           width=[0.02,0.02,0.02], resolution=600,
           item='density', method="integrate")
image.shape
yt.write_projection(image, "projection4.png",
        colorbar_label="Column Density (g cm$^{-2}$)")

# no colourbar, does not automatically pick the log scale
# from numpy import log10
# yt.write_image(log10(image), 'projection4.png')
```

Volume rendering

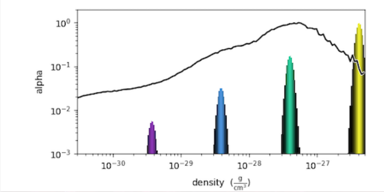
- https://yt-project.org/doc/visualizing/volume_rendering.html
- This is what you see on magazine covers
- Computationally much more demanding
- Currently software ray-tracing
- OpenGL versions “under development”

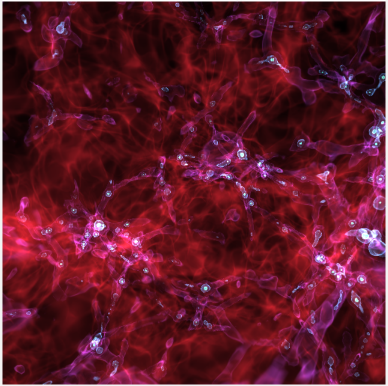
Volume rendering: manual Gaussians




```python
import yt
from numpy import log10
ds = yt.load('/tmp/yt-data/Enzo_64/DD0043/data0043')

# set up a scene for volume rendering
sc = yt.create_scene(ds, lens_type='perspective')

# the first (and the only) source in the scene to be rendered
source = sc[0]
source.set_field('density') # set the source's field to render
source.set_log(True)    # use log (and not linear) space

# start building the transfer function
bounds = (3e-31, 5e-27)
tf = yt.ColorTransferFunction(x_bounds=log10(bounds))
# help(yt.ColorTransferFunction)

# add a red spike [r,g,b,alpha] and then a cyan spike
tf.add_gaussian(location=-28, width=0.003, height=[1,0,0,1])
tf.add_gaussian(location=-29.5,width=0.005,height=[0.5,1,1,0.05])
print(tf)

# apply our transfer function to the source
source.tfh.tf = tf     # tfh stands for TransferFunctionHelper
source.tfh.bounds = bounds
source.tfh.plot('transferFunction.png', profile_field='density')

# save the image, flooring especially bright pixels
sc.save('volume.png', sigma_clip=4)
```

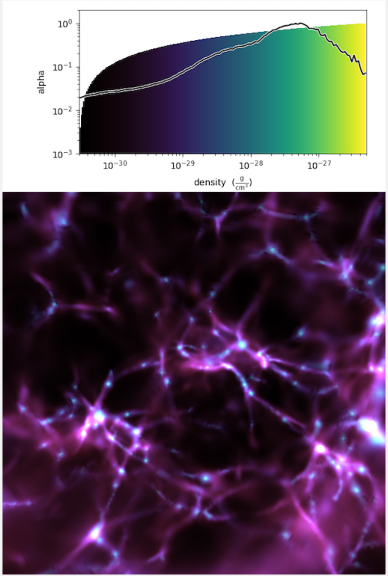
Volume rendering: automatic Gaussians





```
% 18,20c18,19
< # add a red spike [r,g,b,alpha] and then a cyan spike
< tf.add_gaussian(location=-28, width=0.003, height=[1,0,0,1])
< tf.add_gaussian(location=-29.5,width=0.005,height=[0.5,1,1,0.05])
---
> # add multiple evenly-spaced "automatic" Gaussians
> tf.add_layers(N=5, colormap='arbre')

>>> print(tf)
<Color Transfer Function Object>:
x_bounds:[-31, -26] nbins:256 features:
        ('gaussian', 'location(x):-30', 'width(x):0.0039',
                     'height(y):(0.45, 0.077, 0.1, 0.001)')
        ('gaussian', 'location(x):-29', 'width(x):0.0039',
                     'height(y):(0.55, 0.22, 0.72, 0.0056)')
        ('gaussian', 'location(x):-28', 'width(x):0.0039',
                     'height(y):(0.3, 0.58, 0.86, 0.032)')
        ('gaussian', 'location(x):-27', 'width(x):0.0039',
                     'height(y):(0.18, 0.84, 0.63, 0.18)')
        ('gaussian', 'location(x):-26', 'width(x):0.0039',
                     'height(y):(0.95, 0.94, 0.19,   1)')
```

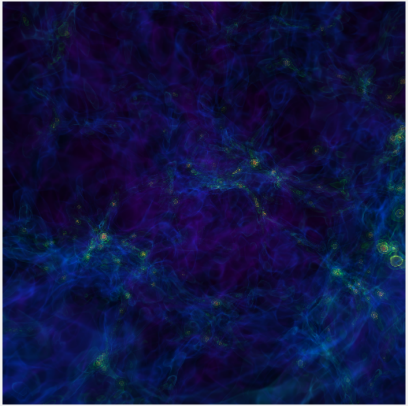
Volume rendering: linear colourmap




```
% 18,19c18,23
< # add multiple evenly-spaced "automatic" Gaussians
< tf.add_layers(N=5, colormap='arbre')
---
> # user-defined function to scale the alpha channel
> def linramp(vals, minval, maxval):
>     return (vals - vals.min())/(vals.max() - vals.min())
> # define a colourmap over a range of densities
> tf.map_to_colormap(mi=log10(3e-31), ma=log10(5e-27),
>                    colormap='arbre', scale_func=linramp)

>>> print(tf)
<Color Transfer Function Object>:
x_bounds:[-31, -26] nbins:256 features:
        ('map_to_colormap', 'start(x):-31', 'stop(x):-26',
         'value(y):  1')
```

Volume rendering: fully automatic

- Let YT do everything by itself: pick the source, compute data bounds, build transfer function



```
import yt
ds = yt.load('/tmp/yt-data/Enzo_64/DD0043/data0043')
sc = yt.create_scene(ds, lens_type='perspective')
sc.save('volume4.png')
```

Moving the scene camera

https://yt-project.org/doc/cookbook/complex_plots.html#cookbook-camera-movement

```
sc = yt.create_scene(ds, ...)
cam = sc.camera

cam.iter_zoom(final, nzoom)    # return an object that produces 'nzoom'
                               # snapshots until the current view has been
                               # zoomed in to 'final' factor

cam.iter_move(center, nmove)    # return an object that produces 'nmove'
                                # snapshots until the current view has been
                                # moved to a final 'center'

cam.iter_rotate(angle, nspin)    # return an object that produces 'nspin'
                                 # snapshots until the current view has been
                                 # rotated by 'angle' radians around the
                                 # vertical axis or the optional 'rot_vector'

sc.save('frame0000.png')    # save an image at the starting orientation
for i in cam.iter_rotate(pi, 30):        # rotate by 180 degrees over 30 frames
    sc.save('frame%04d.png' % (i+1))    # save frame{0001..0030}.png
```

Parallel YT via `mpi4py`

Currently, YT can do the following in parallel:

- slice plots
- projection plots
- off-axis slices
- covering grids (examining grid data in a fixed-resolution array)
- creating and processing derived quantities
- 1D / 2D / 3D profiles and histograms
- halo analysis (halo analysis)
- volume rendering
- isocontours and flux calculations

Installing YT on Cedar and Graham in your directory

```
% # initial setup for CPU rendering
$ cedar
$ module load python
$ virtualenv astro     # install Python tools in your $HOME/astro
$ source ~/astro/bin/activate
$ pip install cython
$ pip install numpy
$ pip install yt
$ pip install mpi4py

# usual use
$ source ~/astro/bin/activate    # load the environment
$ python
...
$ deactivate
```

Rotating a cosmological volume with grids annotations

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

On a cluster, save this as `grids.py`:

```
import yt
from numpy import pi
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)
nspin = 900
for i in cam.iter_rotate(pi, nspin):    # rotate by 180 degrees over nspin frames
    sc.save('frame%04d.png' % (i+1), sigma_clip=4)
```

and this as `yt-mpi.sh`:

```
#!/bin/bash
#SBATCH --time=3:00:00    # 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/astro/bin/activate
srun python grids.py
```

Rotating a cosmological volume with grids annotations (cont.)

- Submit the job

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

- Performance: serial - 1.47 frames/minute, parallel on 4 cores - 4.05 frames/minute
- 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

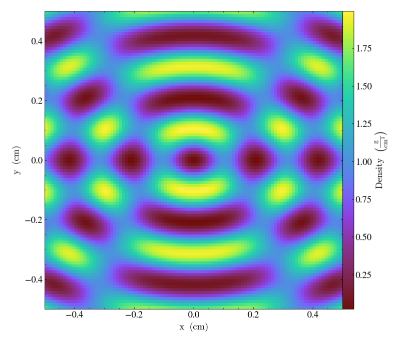
```
$ rsync -av --progress cedar.computecanada.ca:path/to/grids.mp4 .
```

- Final video
  - online (rather compressed) `https://vimeo.com/301503962`
  - on presenter's laptop `grids.mp4`

Generic array data

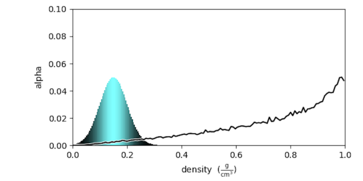
- This is Python $\Rightarrow$ can start with any array in any form
- Let's read from a NetCDF file storing the 3D sine envelope wave function defined inside a unit cube ($x_i \in [0,1]$) at 100^3 resolution

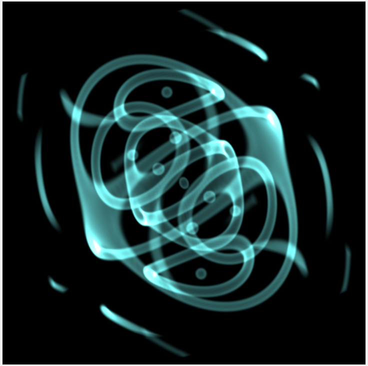
$$f(x_1,x_2,x_3)=\sum_{i=1}^{2}\left[\frac{\sin^2\left(\sqrt{\xi_{i+1}^2+\xi_i^2}\right)-0.5}{\left[0.001(\xi_{i+1}^2+\xi_i^2)+1\right]^2}+0.5\right], \text{ where } \xi_i\equiv 30(x_i-0.5)$$




```
import yt
import numpy as np
from netCDF4 import Dataset
all = Dataset('sineEnvelope.nc', 'r')
rho = all.variables['density'][:,:,:]    # numpy array
data = dict(density = rho)
bbox = np.array([[0,1],[0,1],[0,1]])

# create a yt-native dataset
ds = yt.load_uniform_grid(data=data, domain_dimensions=rho.shape,
                          length_unit=1., bbox=bbox)

slc = yt.SlicePlot(ds, 'z', 'density', center='c')
slc.set_log('density', False)      # use linear colourmap
slc.save('slice7.png')
```

Generic array data (cont.)

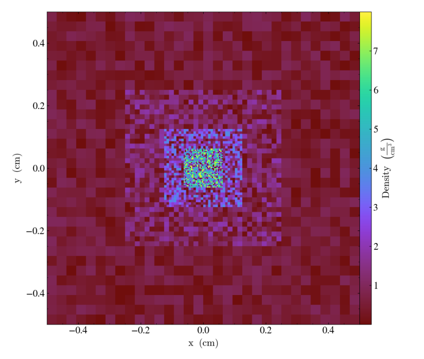




```
sc = yt.create_scene(ds, 'density')
source = sc[0]    # the object in the scene to be rendered
source.set_field('density')    # set the field
source.set_log(False)    # use linear colourmap

# build the transfer function with a cyan spike
bounds = (0,1)
tf = yt.ColorTransferFunction(x_bounds=bounds)
tf.add_gaussian(location=0.15, width=0.005,
                height=[0.5,1,1,0.05])    # [r,g,b,alpha]

# apply our transfer function to the source
source.tfh.tf = tf    # tfh stands for TransferFunctionHelper
source.tfh.bounds = bounds
source.tfh.plot('transferFunction.png', profile_field='density')

sc.camera.resolution = (1024,1024)
sc.camera.position=[1,1,1.5]
sc.save('volume5.png', sigma_clip=3) # remove esp. bright pixels
```

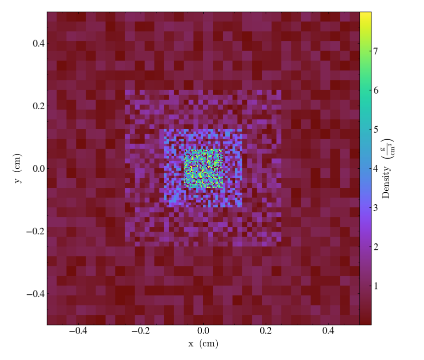
Generic multi-resolution data: set up and plot the grids

- In this toy example let's ignore parent-children relationships between grids
  - YT will attempt to set them, but not necessarily correctly in this case
  - for real AMR patches, we (or the dataset reader function) should specify these relationships explicitly



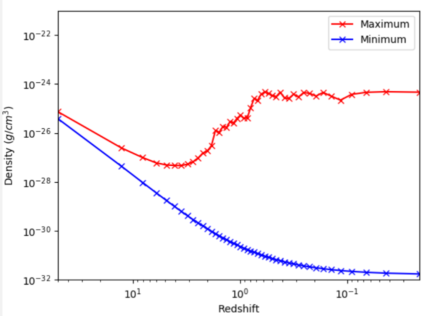

```
import yt
import numpy as np

dims = [32,32,32]
gridData = [
    dict(left_edge=[0.]*3,      right_edge=[1.]*3,
         level=0, dimensions=dims),
    dict(left_edge=[0.25]*3,    right_edge=[0.75]*3,
         level=1, dimensions=dims),
    dict(left_edge=[0.375]*3,   right_edge=[0.625]*3,
         level=2, dimensions=dims),
    dict(left_edge=[0.4375]*3, right_edge=[0.5625]*3,
         level=3, dimensions=dims)]

for g in gridData:
    top = 2**g['level']
    g['density'] = np.random.random(g['dimensions']) * top

ds = yt.load_amr_grids(gridData, domain_dimensions=dims)

slc = yt.SlicePlot(ds, 'z', 'density', center='c')
slc.set_log('density', False)     # use linear colourmap
slc.save('slice8.png')
```

Generic multi-resolution data: examine the grids




```
>>> print(ds.index.num_grids, ds.index.max_level)
4 3

>>> ds.index.grid_levels
array([[0], [1], [2], [3]], dtype=int32)
>>> ds.index.grid_dimensions
array([[32, 32, 32], [32, 32, 32],
       [32, 32, 32], [32, 32, 32]], dtype=int32)
>>> ds.index.grid_left_edge
YTArray([[0., 0., 0.], [0.25, 0.25, 0.25],
   [0.375, 0.375, 0.375], [0.4375, 0.4375, 0.4375]]) code_length
>>> ds.index.grid_right_edge
YTArray([[1., 1., 1.], [0.75, 0.75, 0.75],
   [0.625, 0.625, 0.625], [0.5625, 0.5625, 0.5625]]) code_length

>>> ds.index.grids[0]['density'][0,0,0]
0.49203435485203484 g/cm**3
>>> ds.unit_system['length']
cm
>>> ds.unit_system['density']
g/cm**3

>>> point = ds.point([0.1,0.5,0.5])    # in dataset units
>>> point
YTPoint (AMRGridData): , p=[0.1 0.5 0.5] cm
>>> point['density']    # print density at specific (x,y,z)
YTArray([0.70887875]) g/cm**3
```

Time-series analysis




```
import yt

# load a time sequence DD00{00..46}
ts = yt.load('/tmp/yt-data/enzo_tiny_cosmology/DD????/DD????')

print(len(ts))    # print the number of steps (should be 47)

# scan all time outputs
rho_ex, redshift = [], []
for ds in ts:
    print(ds)     # print basic dataset info
    dd = ds.all_data()    # region = entire simulation domain
    rho_ex.append(dd.quantities.extrema('density'))
    redshift.append(ds.current_redshift)

import numpy as np
rho_ex = np.array(rho_ex)

# plot min/max density vs. redshift
from matplotlib import pylab
pylab.loglog(redshift, rho_ex[:,1], '-xr', label='Maximum')
pylab.loglog(redshift, rho_ex[:,0], '-xb', label='Minimum')
pylab.xlabel('Redshift')
pylab.ylabel('Density ($g/cm^3$)')
pylab.legend()
pylab.xlim(50, 0)
pylab.ylim(1e-32, 1e-21)
pylab.show()
```

Not covered today

- Working with non-astrophysical variable-resolution (curvilinear) and unstructured grids
- Working with particle data
- Bash command-line YT
- 1D / 2D / 3D profiles and histograms
- Data objects – think of them as filters used to define subsets, derivative datasets, collections
  - surfaces (isocontours), streamlines, ...
  - flux calculations
  - subsetting domains and plotting subsets with fixed-resolution buffer
- Intel's EMBREE software ray tracing in YT
- GPU rendering in YT

Questions?