yt Enhancement Proposals Documentation

Release 1.0

The yt Project

May 27, 2022

Contents

This is a repository of "yt Enhancement Proposals" (YTEPs). Because yt is relied upon for production-level science, large-scale design decisions are to be described and discussed before being acted upon. YTEP documents are set up to provide minimal overhead to discussion, while still allowing a clear and evolving specification.

YTEPs go through several stages in their lifetime:

1. Initial writing of the YTEP
2. Announcement and/or discussion on the yt-dev mailing list (may include implementation)
3. Implementation and integration, or possibly declination (this includes pull requests)

YTEPs are useful for any code development that affects how others build tools or infrastructure, and necessary for breaking backwards compatibility. They are not necessary for most new developments or any bug fixes.

Contents:

CHAPTER 1

YTEP-0000: Project Governance

1.1 Abstract

Created: August 24, 2014 Author: Britton Smith Modified: August 09, 2019 Author: Madicken Munk

This document describes the high-level structure, policies, procedures, and processes of the yt project.

1.2 Status

Completed

1.3 Project Management Links

- Apache Software Foundation
- Initial governance discussion on yt-dev: here.
- Secondary discussion about need to update governance from yt-exec: here.

1.4 Detailed Description

1.4.1 Motivation

The yt project consists of a number of repositories within the yt project organization. The organization itself has a number of people interacting with and contributing to these associated repositories. Here we propose a broad overview of the governance model of the project, which will be detailed in a specific governance repository within the yt project, at: https://github.com/yt-project/governance. Major changes to the governance model will be iterated upon here, and details about the model will happen in the governance repository. This will allow small changes within the governance documentation to move quickly and not need to go through a major vote to update the YTEP.

1.4.2 Structure

The governance document will do the following

- Define where the governance documents apply, and how to override them, if relevant
- Provide guidelines on project licensing
- Link to the Code of Conduct, state what happens if a violation occurs, and what avenues are available for reporting violations
- State how conflicts of interest are handled among voting members of the community
- State who the voting members of the community are
- Outline and define the roles within the project, including: contributors, developers, reviewers, and maintainers
- Define how to become a project member, what expections exist for project members, how to become an emeritus member of the project, and how to revoke project membership
- Define how to become a steering committee member, what expections exist for steering committee members, and how members are voted into the steering committee, and how long membership on the steering committee lasts
- Create a project leadership structure that facilites project sustainability, inclusive onboarding practices, and mentorship to learn and understand packages/subpackages within the yt project.
- Make clear guidelines on how voting occurs for changes in the project, including:
  - minor documentation changes
  - code changes and major documentation changes
  - changes to API principles and changes to dependencies or supported versions
  - changes to the project governance
  - project membership.
- State when project meetings happen, at what frequency that they occur, how they are announced to the community, and where they are documented.

1.5 Backwards Compatibility

Sic semper inordinatio.

1.6 Alternatives

The alternative is to continue with no official guidelines and somehow manage, or to continue with an older version of the governance model.

CHAPTER 2

YTEP-0001: IO Chunking

2.1 Abstract

Created: November 26, 2012 Author: Matthew Turk

IO in yt 2.x has always been based on batching IO based on grids. This YTEP describes a new method, which allows for a selection of keywords ('spatial', 'all', 'io') to describe methods of IO that are then left to the frontend or geometry handler to implement. This way, the frontend is able to decide how to access data without any prescriptions on how it should be accessed.

2.2 Status

In-Progress: This has been largely implemented for grid and oct geometries in yt-3.0.

2.3 Project Management Links

- Initial mailing list discussion
- Source of chunking tests

2.4 Detailed Description

2.4.1 Background

"Chunking" in this section refers to the loading of data off disk in bulk. For traditional frontends in yt, this has been in the form of grids: either single or in bulk, grids have been loaded off disk. When Derived Quantities want to handle individual grids, one at a time, they "preload" the data from whatever grids the ParallelAnalysisInterface thinks they deserve. These grids are iterated over, and handled individually, then the result is combined at the end. Profiles do

something similar. However, both of these are de facto, and not really designed. They rely on calls to semi-private functions on data objects, manually masking data, on and on.

An explicit method of data chunking that relies on the characteristics of the desired chunks, rather than the means of the chunking, is needed to bypass this reliance on the grid mediation of IO. In this method, data objects will request that the geometry handler supply a set of chunks. Chunks are of the form (IO_unit, size), where IO_unit is only ever managed or handled by `_read_selection`. This allows the information about all types of IO and collections of data to live internal to the individual implementations of `GeometryHandler` objects. This way, Grids can still batch based on Grid information, but this abstraction is not needed for Octree IO.

Note that YTEP-0017 redefines `GeometryHandler` to `Index` – this reflects the fact that the process of data selection and IO is better thought of as a process of indexing, and that any subsequent operations should be conducted at a higher level.

2.4.2 Main Changes

- Data objects no longer have a `_grids` attribute.
- Parallelism is restructured to iterate over chunks (decided on by the geometry handler) rather than grids
- Grids do not exist outside of the grid geometry handler
- To specifically break backwards compatibility, a `blocks` property has been added which will iterate and yield block-like data (i.e., grids) and the mask of the blocks. This should encompass the use case of both iterating over the `_grids` attribute, obtaining the mask that selects points inside a region, and having a 3D dataset. This should be used exceedingly rarely, but it will be implemented for all types of data. All routines that call upon `_grids` directly must be updated to use `blocks`.

2.4.3 Implementation

The chunking system is implemented in a geometry handler through several functions. The `GeometryHandler` class needs to have the following routines implemented:

- `_identify_base_chunk(self, dobj)`: this routine must set the `_current_chunk` attribute on `dobj` to be equal to a chunk that represents the full selection of data for that data object. This is the "base" chunk from which other chunks will be subselected.
- `_count_selection(self, dobj, sub_objects)`: this must count and return the count of cells within a given data object.
- `_chunk_io(self, dobj)`: this function should yield a series of `YTDataChunk` objects that have been ordered and created to consolidate IO.
- `_chunk_spatial(self, dobj, ngz, sort = None, preload_fields = None)`: this should yield a series of `YTDataChunk` objects which have been created to allow for spatial access of the data. For grids, this means 3D objects, and for Octs the behavior is undefined but should be 3D or possibly a string of 3D objects. This is where ghost zone generation will occur, although that has not yet been implemented. Optionally, the chunk request can also provide a "hint" to the chunking system of which fields will be necessary. This is discussed below.
- `_chunk_all(self, dobj)`: this should yield a single chunk that contains the entire data object.

The only place that `YTDataChunk` objects will ever be directly queried is inside the `_read_fluid_selection` and `_read_particle_selection` routines, which are implemented by the geometry handler itself. This means that the chunks can be completely opaque external to the geometry handlers.

To start the chunks shuffling over the output, the code calls `data_source.chunks(fields, chunking_style)`. Right now only "spatial", "io" and "all" are supported for chunking styles. This corresponds to spatially-oriented division, IO-conserving, and all-at-once (not usually relevant.) The chunks function looks like this:

```
def chunks(self, fields, chunking_style, **kwargs):
    for chunk in self.hierarchy._chunk(self, chunking_style, **kwargs):
        with self._chunked_read(chunk):
            self.get_data(fields)
            yield self
```

Note what it does here – it actually yields *itself*. However, inside the chunked_read function, what happens is that the attributes corresponding to the size, the current data source, and so on, are set by the geometry handler (still called a hierarchy here.) So, for instance, execution might look like this:

```
for ds in my_obj.chunks(["Density"], "spatial"):
    print ds is my_obj
    print ds["Density"].size
```

The first line will actually print True, but the results from the second one will be the size of (for instance) the grid it's currently iterating over. In this way, it becomes much easier to stride over subsets of data. Derived quantities now look like this:

```
chunks = self._data_source.chunks([], chunking_style="io")
for ds in parallel_objects(chunks, -1):
    rv = self.func(ds, *args, **kwargs)
```

It chunks data off disk, evaluates and then stores intermediate results.

This is not meant to replace spatial decomposition in parallel jobs, but it *is* designed to enable much easier and *mesh-neutral* division of labor for parallelism and for IO. If we were to call chunk on an octree, it no longer has to make things look like grids; it just makes them look like flattened arrays (unless you chunk over spatial, which I haven't gotten into yet.)

Essentially, by making the method of subsetting and striding over subsetted data more compartmentalized, the code becomes more clear and more maintainable.

2.5 Field Preloading

A common problem with the current chunking system is the problem of preloading for data access for spatial fields. For instance, inside the field generation system, this construction is used:

```
for io_chunk in self.chunks([], "io"):
    for i,chunk in enumerate(self.chunks(field, "spatial", ngz = 0)):
```

At this point in the system, a single field is being generated and all of the dependencies for that field can be calculated using *_identify_field_dependencies*, but this is not done. The chunking will first break into IO chunks, and then iterate over those chunks in a spatial chunk. This results in IO not being conducted on the IO chunks, but instead on each individual spatial chunk. For octree datasets, this is not typically that bad, as a spatial chunk there can consist of many items. However, for patch-based datasets (particularly Enzo and the current FLASH implementation) this results in far more fine-grained IO access than we want. As an example, this would not allow any batching of IO inside HDF5 files, despite already ordering the access to the spatial data in that appropriate order. When depositing particles in Enzo, for instance, this results in a single access to every single grid for each particle deposition operation.

For non-spatial fields, IO chunking is typically quite effective and appropriate for patch datasets.

To remedy this, we need to construct a language for preloading within an IO chunk. This would necessitate the creation of a `_field_cache` attribute on `DataContainer`, which would be populated *inside* the `_chunk_io` loop, if hinting is available. `_read_fluid_fields` and `_read_particle_fields` would then inspect the chunk they are passed, and for any fields that are requested, if they are inside the `_field_cache` dict (or dict subclass) those values would be returned. This is managed by the `_activate_cache` method.

This would change the loop above to look something like this:

```
field_deps = self._identify_dependencies(field)
for io_chunk in self.chunks([], "io"):
    for i,chunk in enumerate(self.chunks(field, "spatial", ngz = 0,
                                         preload_fields = field_deps)):
```

This should result in much more efficient IO operations as IO for spatial fields will be able to be consolidated. As they are currently implemented, Octrees would likely not need this improvement, and so they will not need to have this implemented. However, all frontends may ultimately benefit from this, as it could trivially be extended to keep all data resident in memory for situations where many passes over a small amount of data are necessary.

2.6 Backwards Compatibility

This system changes how data objects access data, and so this may ultimately result in differences in results (due to floating point error). Additionally, any code that relies on access of the `_grids` attribute on data objects will be broken.

All Octree code will need to be updated for 3.0. All frontends for grids will need to be updated, as this requires somewhat different IO systems to be in place. Updating the grid patch handling will require minimal code change.

Ghost zones have been implemented, but will require further study to ensure that the results are correctly being calculated. Ghost zone-requiring fields are progressing.

To accommodate situations where data objects or processing routines (not derived fields) require information about the shape, connectivity and masking of data, a `blocks` attribute has been implemented. This attribute will yield masks of data and 3D-shaped data containers, enabling most old grids-using routines to work. By focusing on blocks of data rather than grids, we emphasize that these may be of any size, and may also be generated rather than code-inherent data.

2.7 Alternatives

The main alternative for this would be to grid all data, as is done in 2.x. I believe this is not sustainable.

CHAPTER 3

YTEP-0002: Profile Plotter

3.1 Abstract

Created: December 5, 2012 Author: Matthew Turk

This YTEP describes a profile plotting solution, in the style of the PlotWindow, to replace the functionality of the methods on the `PlotCollection` that create profiles. It should have sane defaults, a restricted set of functionality, and should make accessing the underlying matplotlib axes object very easy.

The method proposed needs to meet several competing needs. It should accept objects, it should "do the right thing" for auto-creating profiles, it should provide access to

3.2 Status

Completed

The code can be seen in `yt/visualization/profile_plotter.py`, specifically the objects:

- `PhasePlotter`
- `ProfilePlotter`

3.3 Project Management Links

- Mailing list discussion
- Example notebook

3.4 Detailed Description

3.4.1 Motivation

The `PlotCollection` provides an easy way to quickly make profiles and phase plots, both from spheres that it will create and from objects. However, it suffers from a number of deficiencies:

- Accessing the axes objects from matplotlib is non-trivial
- Plotting multiple profiles on the same axes object is non-trivial
- The profile plotter tries to do too many things, and in doing so does not do anything particularly well
- Profile plots can't be pickled independently of the data

As such, by including this new approach, we will match the functionality from these routines:

- `add_profile_sphere`
- `add_profile_object`
- `add_phase_sphere`
- `add_phase_object`

in a way that will enable access to the underlying plots, pickling of plots, and also easier overplotting and multi-plotting.

3.4.2 Implementation

The implementation will need to:

- Accept a data object and set sane defaults for calculating a profile
- Not add more than one field to the resulting `BinnedProfile` object; only one will be added at any time, and additional fields will be left up to the user
- Include a standalone `Plot`-type object that contains all of the necessary data to create a representation of the data in Matplotlib.
- Deposit the plot into an existing `Axes` or `Figure` object.
- Provide simple methods of translating the initial Profile data into the matplotlib object, and allow the user to modify the visualization as she sees fit.

The current implementation contains this class structure:

`ProfilePlotter`

- `profile`: property, the BinnedProfile1D
- `scale`: "log" or "linear", the scale of the profile's y-axis (x-axis is set by the bins in the profile)
- `_current_field`: property, the name of the field currently selected
- `_setup_plot`: Called to create a new plot specification (*not* a Matplotlib plot) and usually called in the background

PhasePlotter

- `profile`: property, the BinnedProfile2D
- `scale`: "log" or "linear", the scale of the profile's y-axis (x-axis and y-axis are set by the bins in the profile)
- `_current_field`: property, the name of the field currently selected
- `_setup_plot`: Called to create a new plot specification (*not* a Matplotlib plot) and usually called in the background

AxisSpec

This specifies the description (title, bounds, scale, ticks) of an Axis. It is typically used without specifying ticks, which are used only by the ExtJS widgets.

ColorbarSpec

- `cmap`: property, string
- `display`: property, true or false

PlotContainer

Used for profile plots

- `x_spec`: property, AxisSpec
- `y_spec`: property, AxisSpec
- `x_values`: property, numpy array
- `y_values`: property, numpy array
- `to_mpl`: routine that accepts either (Axes), (Figure), or None. Returns the figure and axes after plottin.

ImagePlotContainer

Used for phase plots

- `x_spec`: property, AxisSpec
- `y_spec`: property, AxisSpec
- `image`: property, numpy array (2D)
- `cbar`: property, `ColorbarSpec` instance
- `to_mpl`: routine that accepts either (Axes), (Figure) or None. Returns the figure and axes after plotting.

The indirection enables the user to pickle the plot, without storing the data. But it comes at the price of clarity.

This is currently implemented, and plots returned for the most part to not encourage or allow a substantial amount of modification or fiddling. I think this is okay, as it will very easily allow users to plot multiple lines into the same axes, for instance. However, it does less hand-holding.

3.4.3 Open Questions

- Should the `PhasePlotter` and `ProfilePlotter` objects be refactored to be explicitly favoring matplotlib? The PlotWindow does this, and it is successful.
- Should we allow users to dynamically switch fields on the fly?
- Should the desire for pickling of objects be given up in favor of a cleaner and simpler class structure? (i.e., getting rid of the Specification objects.)

3.5 Backwards Compatibility

We will not remove the existing functionality from `PlotCollection`. So there are no backwards compatibility issues.

3.6 Alternatives

Alternately, we could provide nothing, and encourage users to create their own `BinnedProfile1D` plots. Or, we could provide a much more specific-to-MPL alternative.

CHAPTER 4

YTEP-0003: Standardizing field names

4.1 Abstract

Created: December 11, 2012

Author: Casey Stark, Nathan Goldbaum, Matt Turk

Let's clean up field names in yt, ex "SoundSpeed" -> "sound_speed". The proposed work will serve to remove Enzo-isms and encourage more consistent field names across frontends.

4.2 Status

Completed

4.3 Project Management Links

yt 3.0 goals doc https://docs.google.com/document/d/17Q-rbmTj9PyaTgtN1h6C8vqoWeIZjw_OjFbQp8L3Tkg/edit

Universal field names doc https://docs.google.com/document/d/1Qbt6z27S8VWh8h0kOx–ZYih4BDFAgGlH9pE55I51So/edit

4.4 Detailed Description

4.4.1 Background

The universal field names are PascalCased, while lowercase_underscored names are more standard in Python.

4.4.2 Solution

The field names must be updated to lowercase_underscored format. This means all of the common field names, frontend-specific field names, and all references to field names elsewhere in the code base need to be updated.

Additionally, we will take the opportunity to reorganize the field definitions. Currently all of the field definitions live in the monolithic `universal_fields.py` file, which in yt 2.6 is just shy of 1700 lines long. In the reference implimentation, we have created a new top-level namespace, `yt.fields`. The `fields` module is container for a number of submodules that contain field definitions or macros to create classes of field definitions. Currently, the following submodules are attributes of `yt.fields`:

- **`angular_momentum`** Gas and particle angular momentum fields and field creation macros.
- **`fluid_fields`** Useful fields that are simple functions of the primitive variables in a hydrodynamic simulation. *cell_mass*, *sound_speed*, *pressure*, *entropy*, and *kinetic_energy* all live here.
- **`geometric_fields`** Fields that are functions only of the geometry of the data. This includes *radius*, *x*, *y*, *z*, curvilinear coordinat fields, *zeros*, *ones*, and *grid_level*.
- **`magnetic_fields`** Derived fields useful for analyzing simulation that include magnetic fields.
- **`particle_fields`** Particle derived fields, including all non-local particle deposition fields and local fields that are functions of primitive particle properties.
- **`species_fields`** Fields for chemical species. This is currently a stub.
- **`vector_operations`** Fields that are the gradient, divergence or curl of another field as well as macros for setting up these fields.
- **`universal_fields`** Fields that have so far not been assigned to one of the other field submodules. Eventually this module will be removed once the remaining fields have been reorganized.

Additionally, the `fields` module now contains the logic for field detection as well as the *DerivedField* class.

4.4.3 Testing

The existing unit testing framework as well as an updated set of field detection tests. If there are additional fixes required in untested parts of the code base, we will update them as we find them.

4.5 Backwards Compatibility

We will provide a compatibility layer, allowing yt to map from the old field names to the new ones. This compatibility layer will not be enabled by default.

Right now, one can turn it on via the *_setup_deprecated_fields* function:

```
import yt
ds = yt.load('IsolatedGalaxy/galaxy0030/galaxy0030')
ds._setup_deprecated_fields()
```

In the future we might also enable this via a config parameter.

4.6 Field Names

Naming rules:

- must be lowercase underscored.
- should be as verbose as possible.
- should not include units.
- vector fields are separated into 4 scalar fields with endings _x, _y, _z, and _magnitude.

This is a listing of all field names currently defined in the yt-3.0 universal_fields module and the proposed replacement names.

Current name	Proposal
GridLevel	grid_level
GridIndices	grid_indices
OnesOverDx	ones_over_dx
Ones	ones
CellsPerBin	cells_per_bin
SoundSpeed	sound_speed
RadialMachNumber	radial_mach_number
MachNumber	mach_number
CourantTimeStep	courant_time_step
ParticleVelocityMagnitude	particle_velocity_magnitude
VelocityMagnitude	velocity_magnitude
TangentialOverVelocityMagnitude	tangential_over_velocity_magnitude
Pressure	pressure
Entropy	entropy
sph_r	spherical_r
sph_theta	spherical_theta
sph_phi	spherical_phi
cyl_R	cylindrical_r
cyl_RCode	*remove* (unit duplicate)
cyl_z	cylindrical_z
cyl_theta	cylindrical_theta
DiskAngle	*remove* (replaced by theta)
Height	*remove* (replaced by z)
HeightAU	*remove* (unit duplicate)
cyl_RadialVelocity	cylindrical_radial_velocity
cyl_RadialVelocityABS	cylindrical_radial_velocity_absolute
cyl_RadialVelocityKMS	*remove* (unit duplicate)
cyl_RadialVelocityKMSABS	*remove* (unit duplicate)
cyl_TangentialVelocity	cylindrical_tangential_velocity
cyl_TangentialVelocityABS	cylindrical_tangential_velocity_absolute
cyl_TangentialVelocityKMS	*remove* (unit duplicate)
cyl_TangentialVelocityKMSABS	*remove* (unit duplicate)
DynamicalTime	dynamical_time
JeansMassMsun	jeans_mass (possible unit change)
CellMass	cell_mass
CellMassMsun	*remove* (unit duplicate)
CellMassCode	*remove* (unit duplicate)
TotalMass	total_mass
TotalMassMsun	*remove* (unit duplicate)
StarMassMsun	star_mass (possible unit change)
Matter_Density	matter_density

Continued on next page

Table 1 – continued from previous page

Current name	Proposal
ComovingDensity	comoving_density
Overdensity	overdensity
DensityPerturbation	density_perturbation
Baryon_Overdensity	baryon_overdensity
WeakLensingConvergence	weak_lensing_convergence
CellVolumeCode	*remove* (unit duplicate)
CellVolumeMpc	*remove* (unit duplicate)
CellVolume	cell_volume
ChandraEmissivity	chandra_emissivity
XRayEmissivity	xray_emissivity
SZKinetic	sz_kinetic
SZY	szy
AveragedDensity	averaged_density
DivV	div_v
AbsDivV	div_v_absolute
Contours	contours
tempContours	temp_contours
SpecificAngularMomentumX	specific_angular_momentum_x
SpecificAngularMomentumY	specific_angular_momentum_y
SpecificAngularMomentumZ	specific_angular_momentum_z
AngularMomentumX	angular_momentum_x
AngularMomentumY	angular_momentum_y
AngularMomentumZ	angular_momentum_z
ParticleSpecificAngularMomentumX	particle_specific_angular_momentum_x
ParticleSpecificAngularMomentumY	particle_specific_angular_momentum_y
ParticleSpecificAngularMomentumZ	particle_specific_angular_momentum_z
ParticleSpecificAngularMomentumXKMSMPC	*remove* (unit duplicate)
ParticleSpecificAngularMomentumYKMSMPC	*remove* (unit duplicate)
ParticleSpecificAngularMomentumZKMSMPC	*remove* (unit duplicate)
ParticleAngularMomentumX	particle_angular_momentum_x
ParticleAngularMomentumY	particle_angular_momentum_y
ParticleAngularMomentumZ	particle_angular_momentum_z
ParticleRadius	particle_radius
Radius	radius
RadiusMpc	*remove* (unit duplicate)
ParticleRadiusMpc	*remove* (unit duplicate)
ParticleRadiuskpc	*remove* (unit duplicate)
Radiuskpc	*remove* (unit duplicate)
ParticleRadiuskpch	*remove* (unit duplicate)
Radiuskpch	*remove* (unit duplicate)
ParticleRadiuspc	*remove* (unit duplicate)
Radiuspc	*remove* (unit duplicate)
ParticleRadiusAU	*remove* (unit duplicate)
RadiusAU	*remove* (unit duplicate)
ParticleRadiusCode	*remove* (unit duplicate)
RadiusCode	*remove* (unit duplicate)
RadialVelocity	radial_velocity
RadialVelocityABS	radial_velocity_absolute
RadialVelocityKMS	*remove* (unit duplicate)

Continued on next page

Table 1 – continued from previous page

Current name	Proposal
RadialVelocityKMSABS	*remove* (unit duplicate)
TangentialVelocity	tangential_velocity
CuttingPlaneVelocityX	cutting_plane_velocity_x
CuttingPlaneVelocityY	cutting_plane_velocity_y
CuttingPlaneBX	cutting_plane_bx
CuttingPlaneBy	cutting_plane_by
MeanMolecularWeight	mean_molecular_weight
JeansMassMsun	*remove* (duplicate)
particle_density	particle_density
MagneticEnergy	magnetic_energy
BMagnitude	b_magnitude
PlasmaBeta	plasma_beta
MagneticPressure	magnetic_pressure
BPoloidal	b_poloidal
BToroidal	b_toroidal
BRadial	b_radial
VorticitySquared	vorticity_squared
gradPressureX	grad_pressure_x
gradPressureY	grad_pressure_y
gradPressureZ	grad_pressure_z
gradPressureMagnitude	grad_pressure_magnitude
gradDensityX	grad_density_x
gradDensityY	grad_density_y
gradDensityZ	grad_density_z
gradDensityMagnitude	grad_density_magnitude
BaroclinicVorticityX	baroclinic_vorticity_x
BaroclinicVorticityY	baroclinic_vorticity_y
BaroclinicVorticityZ	baroclinic_vorticity_z
BaroclinicVorticityMagnitude	baroclinic_vorticity_magnitude
VorticityX	vorticity_x
VorticityY	vorticity_y
VorticityZ	vorticity_z
VorticityMagnitude	vorticity_magnitude
VorticityStretchingX	vorticity_stretching_x
VorticityStretchingY	vorticity_stretching_y
VorticityStretchingZ	vorticity_stretching_z
VorticityStretchingMagnitude	vorticity_stretching_magnitude
VorticityGrowthX	vorticity_growth_x
VorticityGrowthY	vorticity_growth_y
VorticityGrowthZ	vorticity_growth_z
VorticityGrowthMagnitude	vorticity_growth_magnitude
VorticityGrowthMagnitudeABS	vorticity_growth_magnitude_absolute
VorticityGrowthTimescale	vorticity_growth_timescale
VorticityRadPressureX	vorticity_radiation_pressure_x
VorticityRadPressureY	vorticity_radiation_pressure_y
VorticityRadPressureZ	vorticity_radiation_pressure_z
VorticityRadPressureMagnitude	vorticity_radiation_pressure_magnitude
VorticityRPGrowthX	vorticity_radiation_pressure_growth_x
VorticityRPGrowthY	vorticity_radiation_pressure_growth_y

Continued on next page

Table 1 – continued from previous page

Current name	Proposal
VorticityRPGrowthZ	vorticity_radiation_pressure_growth_z
VorticityRPGrowthMagnitude	vorticity_radiation_pressure_growth_magnitude
VorticityRPGrowthTimescale	vorticity_radiation_pressure_growth_timescale
x-velocity	velocity_x
y-velocity	velocity_y
z-velocity	velocity_z

4.7 Molecular and Atomic Species Names

Particular care must be taken to name molecular and atomic species in a way that is unambiguous as well as terse. We need to be able to resolve the following types of species:

- CO (Carbon monoxide)
- Co (Cobalt)
- OVI (Oxygen ionized five times)
- H2+ (Molecular Hydrogen ionized once)
- H- (Hydrogen atom with an additional electron)

The naming scheme we have decided upon is of the form `MM[_[mp][NN]]`. `MM` is the molecule, defined as a concatenation of atomic symbols and numbers, with no spaces or underscores. The second sequence is only required if the ionization state is not neutral, and is of the form `p` and `m` to indicate "plus" or "minus" respectively, followed by the number. Our examples above would be `CO`, `Co`, `O_p5`, `H2_p1`, and `H_m1`. Note that we are *not* using an exclusively-lowercase convention here, as we did for the other field names. The name `El` will be reserved for electron fields, as it is unambiguous and will not be utilized elsewhere. Additionally, the isotope of `2H` will be included as `D`.

Neutral ionic species (e.g. H I, O I) are represented as `MM_p0`. For backwards compatibility, neutral species will be mirrored to `MM_` fields, but this practice is deprecated as it is somewhat ambiguous if we are referring to just the neutral species or all atoms of that type. As an example, the neutral hydrogen density field will now be called `H_p0_density`, but it will be mirrored to the `H_density` field for backwards compatibility.

Finally, in those frontends which are single-fluid, we will define these fields for each species:

- `_fraction`
- `_number_density`
- `_density`
- `_mass`

This means that if a frontend only has color fields and species fields (as is the most common case), it will have `("gas", "H2_fraction")` for instance. Otherwise, for multi-fluid calculations (where `gas` is joined by other fields) the other fields will have their own mass and density and so on.

This will require some parsing, and initially we will only support those fields we expect to find. Additionally, because different frontends define these fields in different ways, we will detect which one is the output and define the rest from that. For example, if the frontend finds a `_density` field, the rest will be computed as derived fields from that.

As a point of clarification, the `El_density` currently defined for Enzo is scaled with respect to the ratio of the electron to proton mass ratio. This means that dividing it by `m_h` will result in the number density. Moving forward, this *will not be the case*. We will instead give correct results for mass density when the *alias* is queried. The original name, `Electron_Density`, will still be defined the way it currently is, to preserve access to the original, on-disk fields. (It will be able to be converted to CGS, as well, and will not be scaled in doing so.)

To refer to the number density of the entirety of a single atom or molecule (regardless of its ionization state), please use the `MM_nuclei_density` fields, as opposed to `MM_number_density` fields.

CHAPTER 5

YTEP-0005: Octrees for Fluids and Particles

5.1 Abstract

Created: December 24, 2012 Author: Matthew Turk

In the yt 2.x series, octree AMR codes have largely been supported by re-gridding data to create larger grid patches consisting of both high-resolution data and coarse data. This has the overhead of requiring that each time a dataset (as in from RAMSES or ART) is loaded, the data has to be placed into these grids. This is an expensive process and requires a considerable amount of RAM. This YTEP describes the mechanism in yt 3.0 that directly accesses Octree data, avoiding the costly regridding step and enabling higher-fidelity data access. Additionally, it describes how the Octree data structure will be used for particle data access from datasets such as N-body or SPH simulation output.

5.2 Status

This YTEP is in progress. Most aspects have been implemented in yt 3.0. A major deficiency (described below) is the lack of a distributed memory octree. Discussion of distributed memory Octrees is reserved for a future YTEP.

5.3 Project Management Links

For the most part, this has been conducted internally in the source code.

- Octree data structure

5.4 Detailed Description

Here is where you should write detailed description of what the YTEP proposes. This needs to include:

- Background

- Nature of the problem
- Nature of the solution
- How will the solution be implemented * Brief outline of the code needed to implement this * Code examples of using the solution, in appropriate * How will the solution be tested?
- What are any stumbling points
- What is the proposed method for reaching out to the community about this?

5.4.1 Background

In the 2.x branch of yt, RAMSES and (NMSU) ART data are read and processed in a way that mocks up patch-based AMR data. This is sub-par for several reasons:

1. A costly re-gridding step is required, where octs are deposited into grid patches that are split with some efficiency measure.
2. To conduct IO, coarse grid cells are deposited multiple times into grid patches at finer levels. This results in extremely inefficient IO, as it means that if multiple fine grids overlap with a single cell at coarse resolution, that coarse cell will be read multiple times. It's also very slow.
3. The end result is data that is not exactly what is in the file, reducing the ability of individuals to examine data in a detailed way.
4. The regridding code is difficult to parse and understand, and even harder to extend.

In addition to this, particle codes are simply not available in yt 2.x. All attempts to include them have involved a regridding method similar to that for Octree AMR codes, which is not efficient or high-fidelity. Finally, the RAMSES code is broken in 2.x.

5.4.2 Why is it this way?

The 2.x branch of yt is relatively inflexible in how data is accessed. There are a number of locations that the attributes *grids* or *_grids* are accessed, which are implicitly assumed to be grid patches with a relatively sizable extent. This is used in things like projections, data masking, and the like. For patch-AMR codes where the grids are actually somewhat larger, this is efficient; however, the overhead of python objects and iteration dominate if the grids are smaller than some minimum size or extent. The first implementation of support for RAMSES implemented Octs as grid patches by themselves; this was found to be unbearably slow.

To get around this, a regridding step was applied. This regridding step was based on refinement algorithms, where octs were deposited into grid patches that covered some fraction of the domain. These grid patches were then split to attempt to achieve some minimum efficiency ratio of "refined" (i.e., fine) versus "unrefined" (i.e., coarse) data. When IO was conducted, these were filled in a non-interpolating inverse cascade, where grid patches were filled with fine data, then coarser data. This could be very slow. Additional improvements such as restricting grid patches not to cross "domains" (the RAMSES term for individual files or domains of a specific processr) were eventually added. The NMSU ART data was also loaded in a similar way.

All of this is because yt 2.x relies on "grids" as the fundamental object. As described in *YTEP-0001: IO Chunking*, in yt 3.0 we no longer rely on grids as the object by which all IO is mediated. Data can now be streamed from disk to memory, and coordinates and resolution information can be seen as independent of that data. This allows octrees to exist without a regridding step.

5.4.3 Octree Implementation

The octree implementation is designed around having a full Octree which contains subsets of that octree that are distribute amongst different "domains." The term "domain" comes from RAMSES, and it is best thought of as whatever the natural, IO-oriented subdivision of the data is. For instance, RAMSES divides into multiple files, each of which is called a domain. For purposes of consolidating IO costs, reading on a per-domain basis makes some sense. NMSU ART does not have the concept of multiple domains, and so we can choose to divide data into domains however we like.

Octrees can then be walked to identify which Octs, and then which cells, contribute to a given geometric selector. This can default back to selecting based on the point-by-point location of the Octs, but it can also be queried much more efficiently by early-terminating an octree traversal if a coarse node is not included inside a geometric selector.

This leads nicely to a future where subsets of the octree are not present on every processor; instead, portions can be passed around at will or pinned to specific processors. This is not yet in place, but the Octree has been designed to be forward compatible with this.

For RAMSES data (where the number of Octs is known before any are added to the system), the octree is composed of a set of `OctAllocationContainers`, one for each domain, which are pre-allocated and include all of the Octs themselves. Additionally, there is a base class `OctreeContainer` and a subclass `RAMSESOctreeContainer`. The base class handles and exposes the majority of methods for traversing the octree and querying the octree. The subclass specifies how `Octs` get added to the octree.

Octs are defined to have the following attributes:

- (`np.int64_t`) `ind` – index into the local `OctAllocationContainer`.
- (`np.int64_t`) `local_ind` – index into the global Octree container.
- (`np.int64_t`) `domain` – the domain to which an Oct belongs.
- (`np.int64_t`) `pos[3]` – the integer index, based on the local level's refinement (i.e., the center divided by the local dx)
- (`np.int8_t`) `level` – the level of refinement of the Oct
- (`ParticleArrays`) `*sd` – this is optional, and a pointer to particle arrays. This is typically only used for N-body data and will otherwise be null.
- (`Oct`) `*children[2][2][2]` – Pointers to child nodes. Typically, ifany are null, all are null and the Oct is not refined. However, in ART simulations, the root mesh is defined in cells, rather than octs. This is mocked up in yt as a false mesh of Octs, and so the `children` values can be either NULL (for a refined cell) or not, but may not be homogeneously refined.
- (`Oct`) `*parent` – an upward pointer, for easier traversal of the Octree.

Particle ad N-body data, which does not typically know the organization and structure of the resultant Octree in advance, Uses the additional `ParticleArrays` class for storing particle data that will help govern refinement. `ParticleArrays` have enough data to decide where all of the particles will go during a refinement. This has the downside of mandating that the positions (but no other fields) of all particles in a simulation must, at present, be held in memory. This is a key motivating factor in moving to a distributed octree.

Particle arrays have the following attributes:

- (`Oct`) `*oct` – the Oct to which this particle array belongs.
- (`ParticleArrays`) `*next` – the next particle array in sequence
- (`np.float64_t`) `**pos` – the array of positions for this particle array
- (`np.int64_t`) `*domain_id` – the domain ID (multiple domains mandates refinement in N-body data, as we do not want to span two domains in a single oct.)

- (`np.int64_t`) `np` – the number of particles here.

As noted above there are a number of downsides. Many of these will be simple to fix: for instance, IO right now is characterized by reading in large portions of octrees simultaneously. Furthermore, masks are passed around, although masks are likely an artifact that is no longer necessary (and larget than they need be.)

To add on support for a new Octree code, a subclass of `OctreeContainer` must be made (or `RAMSESOctreeContainer`, if you would like to re-use the `OctAllocationContainer` logic) that implements the following routines:

- `add` – to add new octs to the octree
- `count` – for counting based on a selector
- `icoords`, `ires`, `fcoords`, and IO routines

Additionally, right now the domain subset code is general but not set into base classses. This is also necessary.

5.4.4 Future Work

- Generalize the multi-domain support to allow routines such as `icoords` to be applied generally rather than specifically only for each system of allocation.
- Allow domains to be pinned to processors (distributed memory) and reduce the overhead for individual processors of storing the entire Octree mesh.
- Convert FLASH to use the Octree code.
- Generalize Octree support structures beyond RAMSES.
- Ensure that children can be independently refined.

5.4.5 Stumbling Blocks

1. Spatial data and ghost zones is currently not implemented, and implementation may pose challenes. Part of the reason the implementation for patch-based codes is straightforward is that the arrays come back as 3D arrays, to which (for instance) stencils can be applied. However, for Octree data, we may need to move to returning 4D data to reduce the overhead of processing 10^3 arrays. This means (X,Y,Z,N) where the final dimension is all of the Octs. Retaining compatibility between We also do not want to read outside the domain if not necessary; for instance, RAMSES includes ghost zones in the domain file, even if they are active on a different processor. We should utilize this.
2. Implementation requires a good deal of understanding of how other Octree codes are set up. We should improve readability and make this easier to use.
3. Applying density estimators to particle codes is not yet implemented, and still somewhat unclear. The first implementation will use Voro++ and regions that have some fixed spatial growth affiliated with them. This will likely not be efficient.

5.4.6 Particle Codes

Particle codes are currently supported for reading and creating octree structures. This means that particles can be read in and Octree selection applied to them, where the Octree is refined after either reaching a critical particle count threshold in a given Oct or where an Oct spans multiple domains.

5.5 Backwards Compatibility

Volume rendering no longer works with Octree codes, and will require spatial data support to do so. Additionally, it may be the case that we need to move to a different method for spatial data analysis (X,Y,Z,N) which will require rewriting old scripts.

5.6 Alternatives

I do not believe there are currently credible alternatives to directly understanding Octree data structures in yt. I believe that while we may be able to improve the implemented system, other options such as grid patch conversions are not worthwhile. The particle code support, relying on Octrees for fast selection, could also be implemented using a kD-tree, which may speed the density estimation.

CHAPTER 6

YTEP-0006: Periodicity

Periodicity needs to be dealt with in an explicit, rather than implicit, fashion.

6.1 Abstract

- Created: January 10, 2013
- Modified: January 29, 2013
- Author: Matthew Turk, Nathan Goldbaum

Periodicity in yt has been handled poorly in the past. Some objects and fields have been set to be periodic by default, but not all. This YTEP aims to define a mechanism by which fields and objects can query periodicity information and use it correctly.

6.2 Status

In progress. See pull request #410: http://bitbucket.org/yt_analysis/yt/pull-request/410

6.3 Project Management Links

- Mailing list discussion: http://lists.spacepope.org/pipermail/yt-dev-spacepope.org/2012-December/002739.html
- Mailing list discussion: http://lists.spacepope.org/pipermail/yt-users-spacepope.org/2012-December/003194.html
- Issue: https://bitbucket.org/yt_analysis/yt/issue/484/fields-dont-know-about-periodic-boundary
- Pull request 410: http://bitbucket.org/yt_analysis/yt/pull-request/410

6.4 Detailed Description

Periodicity is a tricky business. By volume, the majority of simulations analyzed with yt are cosmology simulations, which are exclusively periodic. So objects such as ellipsoids, fields such as radius, and so on have all evolved to select data or regard data as wrapping around the edges of simulation boundaries.

However, all of these should only be periodic *when it makes sense*. This means that we need to have a method of marking a simulation as periodic and a method for applying this periodicity. For those situations where periodicity makes sense, it should either *always* be applied if the simulation is periodic, or *never* applied if it is not. I believe we should allow simulations to be periodic in any one of the three axes, but not necessarily all simultaneously; this may be overly complex. We explicitly do not support any type of domain-wrapping or boundary conditions more complex than simply wrapping around.

6.5 Affected Regions

Region of the Code	Type of Periodicity	Change?	Status?
Light cone	Implicit	No	N/A
Halo finding	Implicit	No	N/A
Light ray	Implicit	No	N/A
EnzoFOF	Implicit	No	N/A
FOF	Implicit	No	N/A
Halo objects	Implicit	No	N/A
Fixed Res Buffers	Explicit	Yes	Not done yet
Multi-halo profiler	Implicit	No	N/A
Radial column density	Implicit	Yes	See PR #410
Periodic regions	Explicit	Yes	Not done yet
Spheres	Implicit	Yes	Not done yet
Ellipsoids	Implicit	Yes	Not done yet
Two-point functions	Implicit	Yes	Not done yet
Clumps	Implicit	Yes	Not done yet
Boolean regions	Implicit	Yes	Not done yet
AMR kD Tree	Explicit	Yes	Not done yet
Domain decomp	Implicit	Yes	Not done yet
Radius	Implicit	Yes	Finished: PR 410
ParticleRadius	Implicit	Yes	Finished: PR 410
Covering grids	Implicit	Yes	Not done yet

6.6 New Method

Two types of changes will be made. The first is to remove implicit periodicity and replace it with a check on the periodicity of the simulation. The second is to remove multiple definitions of objects or functions that operate either in periodic or non-periodic methods, and instead provide only one that self-distinguished. Some operations, such as anything that operates on cosmological simulations (which I reluctantly consider halo finding to do) can assume periodicity.

We need to take account of the following types of checks:

- Distance between two points
- Shortest path between two points (uncommon, can be special cased)

- Object inclusion/collision
- Selection of points

We will make the following changes:

- Create a `periodicity` property on all `StaticOutput` objects. This will be a tuple of three booleans, indicating whether or not the simulations are periodic (and if they are, they must be periodic by one domain width). This has been implemented for most of the frontends in PR #410.
- Remove all locally-defined periodicity functions in favor of the function `periodic_dist` in `yt/utilities/math_utils.py` and checking the `periodicity` attribute. For situations where a purely euclidean distance is required, we also supply `euclidean_dist`, which calculates the distance between two points without considering the domain boundaries. These two functions were finalized in PR #410 and currently live in `math_utils.py`.
- Anything that applies periodic shifts to data for checks of inclusion should apply them exclusively through the `periodicity` attribute. For data selectors, we will have a two-step process: the data object will need to implement a `check_periodicity` function.
- Everything that relies on `periodic_region` should instead rely on `region` which will include an option (default = True, which actually means) to check periodicity.
- The `periodic_region` data object will, in 2.X, become a wrapper around the basic region object.

6.7 Backwards Compatibility

All operations that relied on implicit periodicity for datasets that cannot be identified as periodic will have different results.

Old results for non-periodic datasets that were incorrect will become correct.

CHAPTER 7

YTEP-0007: Automatic Pull Requests' validation

7.1 Abstract

Created: February 21, 2013 Updated: January 25, 2015 Author: Kacper Kowalik

This YTEP describes framework used to automatically run both unit and answer testing for incoming pull requests to main YT repository.

Methods proposed here need to be agnostic with respect to chosen continuous integration system.

7.2 Status

Completed

CI server is running at `http://tests.yt-project.org`. Scripts used in validation process are stored in `http://bitbucket.org/xarthisius/yt-validation`.

7.3 Project Management Links

- Mailing list discussion

7.4 Detailed Description

7.4.1 Background

When a new pull request is issued there is no way to automatically test validity of proposed changes. Every author is forced to run the tests manually, which requires: knowledge of how to perform the tests, downloading required data, possessing significant free computing power. This is troublesome and results in tests being run less frequently than

they are supposed to be. With implementation of this YTEP the responsibility of running the testsuite will be shifted from PR's author to designated, automatic infrastructure.

7.4.2 Required Features

The CI infrastructure will need to:

- Constantly poll for incoming changes to main repository (or react to POST message if required API will become available)
- Run both unit and answer testsuite.
- Notify the author should the tests fail.
- Notify the author should the PR could not be cleanly merged.
- Inform people responsible for accepting PR that all tests have passed by sending mail to yt-svn@lists.spacepope.org or broadcasting on #yt irc channel.
- Results of tests should be publicly available.

7.4.3 Implementation Details

Each PR will pulled and merged with current tip in prepared docker container that consists of yt's dependencies: *yt_analysis/devenv*. Resulting container will be tagged as *yt_analysis/yt-PR#:commit_hash* and will be available for download from public docker registry for developers who wish to test changes locally. Subsequently, *yt_analysis/yt-PR#:commit_hash* will be used to run both unit and answer tests on CI server. If PR contains commits that modify or add files to *doc/* subdirectory in the main yt tree, full documentation build will be performed and its result will be stored at `http://tests.yt-project.org`.

Base container *yt_analysis/devenv* will be created using *install_script.sh* and updated every time changes to aforementioned script is merged to yt branch. Additional containers with development environment based on bleeding edge Linux distributions, such as: Gentoo, Debian Sid, Fedora Rawhide, will available and denoted by appropriate tag: *:gentoo*, *:sid*, *:rawhide* respectively. Testsuite will be run on those containers periodically in order to detect incompatibilities with newer versions of yt's dependencies.

Dockerfiles for all containers will be a part of *yt_analysis/yt* repository.

7.4.4 Stumbling Blocks

CI described in this YTEP does not cover integration tests that should be performed on OSX or Windows, nor alternative installations using e.g. wheels, conda. Since CI server is a part of NCSA's infrastructure, access allowing to modify existing and creating new tests will be restricted to people present in NCSA's LDAP.

7.5 Backwards Compatibility

There are no backwards compatibility issues.

CHAPTER 8

YTEP-0008: Release Schedule

8.1 Abstract

Created: February 21, 2013 Author: Matthew Turk Updated: November 3, 2021 Author: Clément Robert

The yt release schedule is somewhat dysfunctional in several ways. Release dates can be difficult to stick to, and merges to stable occur only after long periods. This results in bug fixes not propagating and increases the pressure on developers for a given "release," as each release is seen as monumental rather than incremental. This YTEP describes a new mechanism for increasing the cadence of point releases as well as merging from the development branch into the stable branch.

8.2 Status

Proposed

8.3 Project Management Links

The Mercurial time based release plan, which has partially inspired this discussion, is available here: http://mercurial.selenic.com/wiki/TimeBasedReleasePlan .

8.4 Detailed Description

The yt release schedule is irregular. Here's a table of the releases over time, along with the number of days since the most recent major (i.e., non-point) release. Depending on when this document was last updated, this may include both planned and historical releases.

Version	Release Date	Days Since Last
1.0.1	2008-10-25	N/A
1.5	2009-11-04	375
1.6	2010-01-22	79
1.6.1	2010-02-11	N/A
1.7	2010-06-27	156
2.0	2011-01-17	204
2.0.1	2011-01-20	N/A
2.1	2011-04-06	79
2.2	2011-09-02	149
2.3	2011-12-15	104
2.4	2012-08-02	231
2.5	2013-03-01	211
2.5.1	2013-03-31	30
2.5.2	2013-05-01	30
2.5.3	2013-06-03	33
2.5.4	2013-07-02	29
2.5.5	2013-08-23	52
2.6	2013-11-23	92
2.6.1	2013-12-03	10
2.6.2	2014-02-28	87
2.6.3	2014-07-23	145
3.0	2014-08-04	N/A
3.0.1	2014-09-01	28
3.0.2	2014-10-03	60
3.0.3	2014-11-03	89
3.1	2015-01-14	N/A
3.2	2015-07-24	N/A
3.2.1	2015-09-09	47
3.2.2	2015-11-13	65
3.2.3	2016-02-04	83
3.3.0	N/A	
3.3.1	2016-07-24	171
3.3.2	2016-10-26	94
3.3.3	2016-12-12	47
3.3.4	2017-02-13	63
3.3.5	2017-03-08	23
3.4	2017-08-11	156
3.4.1	2018-02-16	189
3.5	2018-10-16	242
3.5.1	2019-02-26	133
3.6	2020-04-11	410
3.6.1	2020-11-13	216
4.0	2021-07-06	235
4.0.1	2021-07-21	15
4.0.2	2022-02-01	195

In principle, a long release schedule is not a problem. However, what this results in is a reluctance to merge to the stable branch. This has two major side effects: it leads to many people working off of the development branch and it leads to a long time between bug fixes for individuals working off of the stable branch. The development branch, despite its name, is quite stable – however, this also means that when instabilities (or API changes) are introduced in the development branch, it can be much more disruptive.

8.4.1 What Constitutes a Release

The majority of development in the primary yt repository is stable. Seldom are backwards-incompatible changes introduced, nor functionality broken. This is helped by continuous integration and detailed code review. As such, for the most part, yt is in a constant state of "release."

For the purposes of this document, a "release" constitutes five things:

- A new build of the documentation with API and cookbook is placed in a long-term container.
- The development branch (`yt`) is merged to the stable branch (`stable`)
- A new tag in the version control history
- An upload of the source code to PyPI (https://pypi.org/)
- An new entry on conda-forge (https://anaconda.org/conda-forge/yt)
- An announcement email (to `yt-users` for minor releases and more broadly for major releases)
- For "bugfix"-level releases, changes should be backported to a dedicated branch.

8.4.2 Release Managers

The release manager for minor releases will be Matthew Turk, as they will only be announced to `yt-users`. For major releases, a new release manager will be selected by consensus in the `yt-dev` community. Merging, tagging and uploading will be handled by Matthew Turk, but the release manager will act as "whip" to ensure the necessary documentation building is done. Additionally, this release manager will write the release notes and send the email to various mailing lists.

Version	Release Manager
2.5	John ZuHone
2.5.1	Matthew Turk
2.5.2	Matthew Turk
2.5.3	Matthew Turk
2.6	Kacper Kowalik
2.6.1	Matthew Turk
2.6.2	Matthew Turk
2.6.3	Matthew Turk
3.0	Matthew Turk
3.1	John Zuhone
3.2	Britton Smith
3.2.1	Nathan Goldbaum
3.2.2	Nathan Goldbaum
3.2.3	Nathan Goldbaum
3.3.0	Nathan Goldbaum
3.3.1	Nathan Goldbaum
3.3.2	Nathan Goldbaum
3.3.3	Nathan Goldbaum
3.3.4	Nathan Goldbaum
3.3.5	Nathan Goldbaum
3.4.0	Nathan Goldbaum
3.4.1	Nathan Goldbaum
3.5.0	Nathan Goldbaum
3.5.1	Nathan Goldbaum
3.6.0	Madicken Munk
3.6.1	Madicken Munk
4.0.0	Madicken Munk
4.0.1	Madicken Munk
4.0.2	Matthew Turk

8.5 Backwards Compatibility

This should have no backwards-incompatible changes.

8.6 Alternatives

One alternative would be to forego release numbers and move to completely continuous integration. Another would be to continue on our current path.

CHAPTER 9

YTEP-0009: AMRKDTree for Data Sources

9.1 Abstract

Created: February 28, 2012 Author: Sam Skillman

This proposal outlines the changes (functional and API) necessary for the ability to render volumes using arbitrary data sources. This will still operate with the idea of grids and masks. However, this should lead as a stepping stone to non-grid-based rendering.

9.2 Status

Status should be one of the following:

1. Implemented in yt-3.0 PR77

YTEPs do not need to pass through every stage.

9.3 Project Management Links

Currently the development of this capability is in the camera-refactor bookmark at: https://bitbucket.org/samskillman/yt/commits/all/tip/..bookmark%28%22camera-refactor%22%29

There is a Camera refactor YTEP-0010 that is closely related to the AMRKDTree functional changes suggested in this YTEP.

This has been implemented as of: https://bitbucket.org/yt_analysis/yt-3.0/pull-request/77/add-legitimate-source-rendering-using-a/diff

9.4 Detailed Description

Functional Background:

The volume rendering has long operated on either the entire domain or (at best) a sub-rectangular-prism using left and right edges. This is fairly limiting in that a user may not need (or want) to render the entire volume, and may want to restrict the volume shown by something other than a single box. A majority of the reasoning behind the recent `AMRKDTree` was to allow for more generic adding of grids/data to the homogenized volume.

The primary problem with attempting to do this is that the volume rendering acts on a rectangular brick of data, and the traversal of this brick is fairly far removed from a AMR3DData object. Therefore, we either need to modify the traversal, or somehow mask out the data being handed to the traversal.

The latter approach can be accomplished using the following code:

```
def _source_mask(field, data):
    return 1.0*self.source._get_cut_mask(data)
self.pf.field_info.add_field('source_mask', function=_source_mask, take_log=False)
```

then when creating the vertex centered data:

```
mask = grid.get_vertex_centered_data('source_mask',smoothed=False,no_ghost=self.no_
↪ghost).astype('float64')
mask = np.clip(mask, 0.0, 1.0)
mask[mask<1.0] = np.inf
for i,field in enumerate(self.fields):
    vcd = grid.get_vertex_centered_data(field,smoothed=True,no_ghost=self.no_ghost).
↪astype('float64')
    vcd = vcd*mask
```

However, this approach is full of quirks since what we really want is to mask out an entire cell, and not some set of vertices. Therefore, we propose to modify the `PartitionedGrid` object to include an integer mask that is used during traversal to mask out the individual cells.

API Background:

The method for specifying a data source in a volume rendering has not been suggested. Currently, the rectangular volume that is used for rendering uses a `le` and `re` pair of keywords to specify the left and right edges. When moving to a data source, we should simplify the volume selection by simply supplying a `data_source=` keyword.

Changes to the Camera interfaces suggested in YTEP-0010 will handle the move to using a data_source.

In my working solution, I have set the AMRKDTree __init__ function to have the following form:

```
class AMRKDTree(ParallelAnalysisInterface)
    def __init__(self, pf, min_level=None, max_level=None, data_source=None):
```

9.5 Backwards Compatibility

This YTEP breaks the following backwards compatibility:

- AMRKDTree API

It will additionally break internal uses of the API for the Camera, other cameras inheriting the __init__ of Camera, and the AMRKDTree.

9.6 Alternatives

- Do nothing
- Add more keyword arguments to everything
- Wait until rendering is ready in yt-3.0, which will also likely demand a breakage of API.

After discussion, it was found to be easiest to only implement in yt-3.0, as it is increasingly difficult to manage the two versions and how they handle grids. Since in yt-3.0 there are explicit mask objects in the pf.h.blocks generator, it was significantly easier than expected to implement the mask on the PartitionedGrid object. I'm also hopeful that this simplification is along the same lines of the idea of a simplified volume rendering scene object, as outlined in YTEP-0010.

CHAPTER 10

YTEP-0010: Refactoring for Volume Rendering and Movie Generation

10.1 Abstract

Created: March 3, 2013 Author: Cameron Hummels

This YTEP describes significant modifications of the camera infrastructure to enable more focus on scenes, camera paths, and movies, while still retaining functionality for individual images.

10.2 Status

Open to changes through pull requests.

10.3 Project Management Links

This integrates directly with YTEP-0009 currently in pull request at: https://bitbucket.org/yt_analysis/ytep/pull-request/11/data-source-rendering-camera-refactor

10.4 Detailed Description

10.4.1 Background

Visualization of data is one of the primary reasons why people use yt. yt's visualization capabilities are quite advanced, particularly in generating single images of a simulated volume. However, the tools for using the camera objects are complicated and difficult to use to generate movies of anything beyond simple camera paths around single simulation outputs. This is understandable based on the way the camera object code built up organically over the last few years, but a refactor of this code could dramatically simplify the steps for generating complex movies.

Here is a rough algorithm of how to create a rendering under the current system:

1. The user chooses a method for breaking up the region to be rendered, either a Homogenized Volume, or a kd-tree. Homogenized volumes can be re-used from rendering to rendering, thereby saving time in re-rendering the same volume from different perspectives, as well as allowing the user to define an arbitrary geometric object to act as the rendering volume. On the other hand, the kd-tree is generally faster for individual renderings, but cannot currently be re-used from rendering to rendering, and does not allow the user to specify a subregion to render beyond an AMRRegion object (i.e. box).
2. The user must explicitly choose whether she wants to do a volume rendering , or if she wants to do an off-axis projection, as each of these two options is a different camera class. There can be no mixing between these classes–once this is chosen, the user is locked in.
3. The user must explicitly define the central focus point of the image to be rendered, along with the 'width' of the image (thereby defining the extent to be rendered), the normal vector of the from which the camera will render, and the resulting resolution of the final image.
4. The user must explicitly define a transfer function to be used in the case of the volume rendering, and it is generally non-intuitive as to how to get this correct a priori.

Here is a sample script for generating a volume rendering under the current system taken from the docs. Note how much has to be done prior to actually rendering an output image.

```
>>> from yt.mods import *
>>> pf = load("Enzo_64/DD0043/data0043")
>>> dd = pf.h.all_data()
>>> mi, ma = dd.quantities["Extrema"]("Density")[0]
>>> tf = ColorTransferFunction((np.log10(mi)+1, np.log10(ma)))
>>> tf.add_layers(5, w=0.02, colormap="spectral")
>>> c = [0.5, 0.5, 0.5]
>>> L = [0.5, 0.2, 0.7]
>>> W = 1.0
>>> Npixels = 512
>>> cam = pf.h.camera(c, L, W, Npixels, tf)
>>> cam.snapshot("%s_volume_rendered.png" % pf, clip_ratio=8.0)
```

10.4.2 Problem

Here we note some of the shortcomings of the current camera implementation:

1. Too much overhead in generating a simple rendering for the end user. Needs helper functions to use sensible defaults so the user must only call one or two commands before generating a rendering.
2. Too much complexity in the camera object constructor due to a large number of parameters and keyword options.
3. Volume renderings and off-axis projections are too distinct from each other.
4. Not enough focus on time series or persistence of a camera from from one rendering to another for generating movies.
5. Cameras are defined in a somewhat counterintuitive manner, rather than focusing on the camera as being in a physical location in a volume and moving it around that volume.
6. Not enough integration of particle data in camera renderings.
7. Minimal ability to move a camera in a complex path through a volume beyond simple rotations, pans, and zooms.

10.4.3 Proposed Solution

We propose to break up the camera infrastructure into a few different classes to enable more transparency and usability of this important functionality.

- Make cameras just cameras. They should be very lightweight, should be situated in a scene, and should not contain references to the volumes.
- Add a "scene" object which then contains components like data sources (i.e. volumes, streamlines, particles), cameras, transfer functions, etc. The scene remains a structure for modifying the underlying components in that scene throughout the duration of the scene.
- Make a camera a reusable object for a given movie which can be modified in virtually any way (location, transfer function, underlying pf) through a series callback functions, or modifying the scene object directly.
- Remove the homogenized volume method for generating volume renderings and make the kd-tree method handle all functionality that homogenized volumes provided (e.g. reusability, usability on an arbitrary geometric object – see ytep 0009).
- Integrate all current camera classes into a single camera class, so we don't have separate classes for volume renderings, projections, stereoscopic renderings, HEALpix renderings, etc.
- Make the scene understand how to traverse from point A to point B in a complex way by designating keyframes where you constrain the exact rendered image (position/orientation of camera, state of transfer function, data source for rendering, etc.) and having the scene figure out a smooth transition between these keyframes.
- Remove a ridiculous amount of complexity from the Camera and Volume objects by stripping out a large number of variables from the constructors.
- KDTrees should be built for the volume active at any time for easy reusability in future frames (e.g. by moving the camera or changing the transfer function). If the underlying data source changes, then the old kdtree is purged and a new one for that new data source is constructed. This will dramatically reduce overhead on rendering the same volume from different perspectives.
- By default, when one defines a Scene object from a single datadump, it sets the Timeline object to 1 output frame, whereas if one defines a Scene object from a TimeSeries, it adds keyframes for each pf in that TimeSeries uniformly across the Timeline object.

In short, we propose that by reducing complexity of individual objects and splitting them in to multiple objects, we can reduce the complexity of individual operations by adding in a slightly larger set of objects that are more flexible.

New classes:

- **Scene** Meant to be the main class for dealing with volumetric visualization. It is constructed using a static output instance, which it uses to set up a default camera based on domain extents. It also instantiates a list of objects to be rendered, which include RenderSource instances for volume rendering and streamlines.
- **RenderSource** Base class for rendering types. This can be (minimally) volumetric data for volume rendering, path data for streamlines, point data for particles, and other yet to be determined data types.
- **Camera** A lightweight camera representing the location and orientation of the camera. This can be specified in a number of ways, but to uniquely define it, we need position of camera, pointing vector, and an optional north vector (which is used to determine the image "up" direction which specifies the image "up" direction).
- **Timeline** The timeline object represents how the scene changes with time. It is valid from t=0 to t=1, but this can be mapped on to any number of output frames during the render. One can modify the Timeline object by setting events such as keyframes to change the underlying scene components at any point in the timeline.

- **CameraPath** In dealing with movies, one can set key frames of where and in what orientation one wants the camera to be at certain times. A smoothing function (like a spline) can connect up these keyframes into a smooth camera path for application on the timeline.

In each of these following derived classes, the returned object from the __init__ function is an instance of the Scene class, capable of adding additional sources. These are meant to provide shortcuts

Derived Classes:

- **VolumeScene** Inherits from Scene, sets up a scene with a volume rendering data source
- **StreamlineScene** Inherits from Scene, sets up a scene with streamlines data source
- **ParticleScene** Inherits from Scene, sets up a scene with particles data source

10.4.4 Sample Scripts for Proposed Infrastructure

Under the proposed changes, one could simply get a simple volume rendering by running this short script:

```
>>> from yt.mods import *
>>> pf = load("Enzo_64/DD0043/data0043")
>>> sc = VolumeScene(pf, 'Density')
>>> im = sc.render()
```

where the scene constructor uses helper functions to set up all of the default objects (volume, camera, timeline, transfer function) in order to use the entire volume, place a camera at *1.5*domain_right_edge* pointing at *domain_center* and north vector (nx,ny,nz)=(0,0,1), make the timeline object number_of_frames=1, setting the transfer function to use the min/max of the volume and adding 4 isodensity contours.

The previous image can be grabbed using:

```
>>> im = sc.current_image
```

If one wanted to modify this scene prior to rendering, a series would allow the end user to change things through a series of callbacks:

```
>>> from yt.mods import *
>>> pf = load("Enzo_64/DD0043/data0043")
>>> sp = pf.h.sphere([0.5,0.5,0.5],100/pf['kpccm'])
>>> sc = VolumeRender(sp, 'Density')
### Change the camera position and orientation
>>> sc.camera.move(pos=[0,(100,'kpccm'),0], focus=[0,0,0], north=[0,0,1])
>>> sc.render()
```

In order to create a short movie making a rotation around the center from one side at 100 kpc out to the other side 100 kpc out while the simulation is evolving, one might run a script such as the following. It would automatically set the timeline to match the timeseries data with a framerate of 12 frames/sec.

```
>>> from yt.mods import *
>>> ts = TimeSeriesData.from_filenames("Enzo_64/DD????/data????")
>>> sc = scene(ts)
>>> keyframe_start = camera(pos = [0,1,0], point = [0,0,0], north = [0,0,1])
>>> keyframe_mid = camera(pos = [1,0,0], point = [0,0,0], north = [0,0,1])
>>> keyframe_end = camera(pos = [0,-1,0], point = [0,0,0], north = [0,0,1])
>>> sc.set_keyframe(time=0, camera = keyframe_start)
>>> sc.set_keyframe(time=0.5, camera = keyframe_mid)
>>> sc.set_keyframe(time=1, camera = keyframe_end)
```

(continues on next page)

(continued from previous page)

```
>>> sc.timeline.set_num_frames(50)
>>> sc.render()
```

While all the prior examples are focused on handling a single data source at a time, a major goal of the refactor is to allow for the combination of data sources and data types, such as streamlines, particles, opaque planes, and annotations. We want to allow for the composure of a full scene containing many different sources.

For example,

```
>>> from yt.mods import *
>>> pf = load("Enzo_64/DD0043/data0043")
>>> sc = Scene(pf)
### Change the rendered volume to be a sphere of radius 100 kpc
>>> sp = pf.h.sphere([0.5,0.5,0.5],100/pf['kpccm'])
>>> vr_handle = sc.add_volume_rendering(sp)
### Here vr_handle is an instance of a VolumeRenderSource(RenderSource)
>>> vr_handle.transfer_function.clear()
>>> vr_handle.transfer_function.map_to_colormap(mi, ma, cmap='RdBu')
>>> streamlines = Streamlines(pf,...) # Create streamlines
>>> stream_handle = sc.add_streamlines(streamlines)
>>> stream_handle.set_opacity(0.1)
>>> stream_handle.set_radius((0.1,'kpc'))
>>> sc.add_particles(sp)
>>> particle_handler = sc.get_particle_handle()
>>> particle_handler.transfer_function.set_color_field('density')
>>> particle_handler.transfer_function.set_alpha(0.1)
>>> sc.render()
### Remove piece of the scene
>>> sc.toggle(vr_handle)
... # Type                          Tag    Status
... VolumeRenderSource(density): vr_1   off
... Streamlines(velocity):       sl_1   on
... Particles(density):          pt_1   on
>>> sc.render()
>>> sc
... # Type                          Tag    Status
... VolumeRenderSource(density): vr_1   off
... Streamlines(velocity):       sl_1   on
... Particles(density):          pt_1   on
>>> sc.toggle('vr_1')
... # Type                          Tag    Status
... VolumeRenderSource(density): vr_1   on
... Streamlines(velocity):       sl_1   on
... Particles(density):          pt_1   on
```

10.5 What Needs to be Decided

- What should the syntax be for annotations (lines, boxes, orientation vectors)?
- How do we manipulate the Scene positions (positions of all non-spatial annotations)? For example, put the transfer function display over here.
- Probably many more things.
- How to handle the API of running in different parallel regimes (Image plane vs domain vs time-series vs . . .)

10.6 Backwards Compatibility

This will break all backwards compatibility with the pf.h.camera interface. We will attempt to keep as many of the useful modifications (pitch, roll, yaw, etc.) as similar as possible to ease the pain.

CHAPTER 11

YTEP-0011: Symbol units in yt

11.1 Abstract

Created: March 7, 2013
Authors: Nathan Goldbaum, Casey Stark, Anna Rosen, Matt Turk

This YTEP describes adding symbolic units to `yt` using the sympy package. The main benefit is to make sure units are carried through calculations in a *transparent* and *intuitive* manner. The new components are:

- a `Unit` class, which describes the dimensionality (powers of mass, length, time, temperature) and conversion factor of any unit.
- a `UnitRegistry` class, which stores valid atomic unit symbols (ex: "g" for gram).
- a `YTArray` class, a subclass of NumPy `ndarray` that attaches a `Unit` object.
- a `YTQuantity` class, a subclass of *YTArray* which is restricted to a single element (for handling scalars).

11.2 Status

Completed

11.3 Project Management Links

The code can be found in the unit refactor pull request:

https://bitbucket.org/yt_analysis/yt/pull-request/662/unit-refactor/diff#comment-897255

The latest work is in Matt Turk's fork in `yt` in the `unitrefactor` bookmark:

https://bitbucket.org/MatthewTurk/yt/commits/branch/unitrefactor

The work is based on Casey's dimensionful library:

https://github.com/caseywstark/dimensionful

11.4 Detailed Description

11.4.1 Background

The current system for units is functional but not terribly flexible. All data are treated as scalars and it is up to the user to convert data from CGS, which `yt` uses internally, to their chosen unit system. A sample workflow might look like this:

```
from yt.mods import *
from yt.utilities.physical_constants import mass_sun_cgs
pf = load('IsolatedGalaxy/galaxy0030/galaxy0030')
dd = pf.h.all_data()
mass = dd['CellMass']
print "Mass in CGS: ", mass
print "Mass in Solar Masses: ", dd['CellMass']/mass_sun_cgs
print "Mass in code units: ", dd['CellMass']/pf['Mass']
```

This model works well if a user always uses CGS units. If the user needs a quantity in a different unit system, they run into trouble. This is illustrated above in the example to convert to 'solar mass' units, since this isn't a proper unit, the conversion isn't stored inside the *pf* dict, so a user will either need to import the unit definition from yt, or add their own definition to their script. The situation is a little bit better for length conversions:

```
dx = dd['dx']
print "Cell dx in code units: ", dx
print "Cell dx in centimeters: ", dx*pf['Length']
print "Cell dx in megaparsecs: ", dx*pf['mpc']
```

This works pretty nicely, since all of the various length units are stored in the *pf* dictionary. However, this example illustrates another problem; here *dx* is returned in code units, while most quantities are returned in CGS. If we wanted to enforce that all quantities be returned in CGS, we would need to painstakingly go through the codebase, tweaking the field definitions and places where fields are used so that units are handled properly. Clearly, a better solution is needed.

Cosmological units are also handled in a somewhat ad hoc way. Each of the code frontends need to detect that a simulation was performed using comoving units, and define new scaled, comoving and scaled comoving units (i.e. 'kpccm', 'kpchcm' and 'kpch'). This encourages duplication of code in each of the frontends and makes likely that different frontends will ignore some of the cosmological units that are defined in the Enzo frontend. In addition, this is not documented in the frontend docs, making it easier for newly supported codes to miss this. Cosmological units are also not labeled correctly in plots.

To ensure units to display nicely on plots, the unit definition is currently encoded as a raw string in LaTeX format:

```
add_field("MagneticEnergy",function=_MagneticEnergy,
          units=r"\rm{ergs}\/\rm{cm}^{-3}",
          display_name=r"\rm{Magnetic}\/\rm{Energy}")
```

This is harmful for readability and has the effect that user-defined or automatically generated fields are not assigned units.

11.4.2 Proposed Solution

We propose to handle units in a more automatic fashion, leveraging the symbolic math library sympy. Instead of returning a NumPy `ndarray` when users query for fields, the `__getitem__` selector on data objects will return a `YTArray`, a subclass of `ndarray`. This preserves `ndarray`'s array operations, including deep and shallow copies, broadcasting, and views.

Additonally, `YTArray` has a `Unit` object attached to it that tracks units associated with each value in the array. This is encoded in the `__repr__` method of `YTArray`:

```
>>> dd['density']
YTArray([  4.92775113e-31,   4.94005233e-31,   4.93824694e-31, ...,
           1.12879234e-25,   1.59561490e-25,   1.09824903e-24]) g/cm**3
```

`YTArray` defines several user-visible member functions:

- `convert_to_units`

  Converts an array to a valid unit, specified by a string argument. Valid units possess the same `dimension` expression as the current unit.

- `convert_to_cgs`

  Converts the array to CGS units.

- `in_units`

  Returns a copy of the array in a valid unit, specified by a string argument. Valid units possess the same `dimension` expression as the current unit.

- `in_cgs`

  Returns a copy of the array in CGS units.

It's important to remember that `convert_to_cgs` and `convert_to_units` do in-place conversion of an existing array and `in_units` and `in_cgs` return a copy of the original array in the new unit. This can get complicated if one isn't careful about the distinction between creating copies and references, as illustrated in the following example:

```
>>> dens = dd['density']
>>> print dens
[  4.92775113e-31   4.94005233e-31   4.93824694e-31 ...,   1.12879234e-25
   1.59561490e-25   1.09824903e-24] g/cm**3
```

```
>>> dens.convert_to_units('Msun/pc**3')
>>> print dens
[  7.27920765e-09   7.29737882e-09   7.29471191e-09 ...,   1.66743685e-03
   2.35702085e-03   1.62231868e-02] Msun/pc**3
```

```
>>> dd['density'].in_units('Msun/pc**3')
YTArray([  7.27920765e-09,   7.29737882e-09,   7.29471191e-09, ...,
           1.66743685e-03,   2.35702085e-03,   1.62231868e-02]) Msun/pc**3
```

In the example above, if a user tries to query `dd['density']` again, they will find that it has been converted to solar masses per cubic parsec, since a shallow copy, `dens`, underwent an in-place unit conversion. In practice this is not a big concern, since the unit metadata is preserved and the array values are still correct in the new unit system, all numerical operations will still be correct.

One of the nicest aspects of this new unit system is that the symbolic algebra for unitful operations is performed automatically by sympy:

```
>>> print dd['cell_mass']/dd['cell_volume']
[  4.92775113e-31   4.94005233e-31   4.93824694e-31 ...,   1.12879234e-25
   1.59561490e-25   1.09824903e-24] g/cm**3
```

```
>>> print dd['density']
[  4.92775113e-31   4.94005233e-31   4.93824694e-31 ...,   1.12879234e-25
   1.59561490e-25   1.09824903e-24] g/cm**3
```

YTArray is primarily useful for attaching units to NumPy `ndarray` instances. For scalar data, we have created the new `YTQuantity` class. In the proposed implementation, `YTQuantity` is a subclass of `YTArray` with the requirement that it is limited to one element. `YTQuantity` is primarily useful for physical constants and ensures that the units are propogated correctly when composing quantities from arrays, physical constants, and unitless scalars:

```
>>> from yt.utilities.physical_constants import boltzmann_constant
>>> print dd['temperature']*boltzmann_constant
[  1.28901607e-12   1.29145540e-12   1.29077208e-12 ...,   1.63255263e-12
   1.59992074e-12   1.40453862e-12] erg
```

With this new capability, we will have no need for fields defined only to handle different units (e.g. `RadiusCode`, `Radiuspc`, etc.). Instead, there will only be one definition and if a user needs the field in a different unit system, they can quickly convert using `convert_to_units` or `in_units`.

When a `StaticOutput` object is instantiated, it will its self instantiate and set up a `UnitRegistry` class that contains a full set of units that are defined for the simulation. This is particularly useful for cosmological simulations, since it makes it easy to ensure cosmological units are defined automatically.

The new unit systems lets us to encode the simulation coordinate system and scaling to physical coordinates directly into the unit system. We do this via "code units".

Every `StaticOutput` object will have a `length_unit`, `time_unit`, `mass_unit`, and `velocity_unit` attribute that the user can quickly and easily query to discover the base units of the simulation. For example:

```
>>> from yt.mods import *
>>> ds = load("Enzo_64/DD0043/data0043")
>>> print ds.length_unit
128 Mpccm/h
```

Additionally, we will allow conversions to coordinates int the simulation coordinate system defined by the on-disk data. Data in code units will be available by converting to `code_length`, `code_mass`, `code_time`, `code_velocity`, or any combination of those units. Code units will preserve dimensionality: an array or quantity that has units of `cm` will be convertible to `code_length`, but not to `code_mass`.

On-disk data will also be available to the user, presented in unconverted code units. To obtain on-disk data, a user need only query a data object using an on-disk field name:

```
>>> from yt.mods import *
>>> ds = load("Enzo_64"/DD0043/data0043")
>>> dd = ds.h.all_data()
>>> print dd['Density']
[  6.74992726e-02   6.12111635e-02   8.92988636e-02 ...,   9.09875931e+01
   5.66932465e+01   4.27780263e+01] code_mass/code_length**3
>>> print dd['density']
[  1.92588950e-31   1.74647714e-31   2.54787551e-31 ...,   2.59605835e-28
   1.61757192e-28   1.22054281e-28] g/cm**3
```

Here, the first data object query is returned in code units, while the second is returned in CGS. This is because `Density` is an on-disk field, while `density` is a 'standard' `yt` field. See *YTEP-0003: Standardizing field names*.

Unit labels for plots will be programatically generated. This will leverage the sympy LaTeX output module. Even though the field definitions will have their units encoded in plain text, we will be able to automatically generate LaTeX to supply to matplotlib's mathtext parser.

11.4.3 Implementation

Our unit system has 6 base dimensions, `mass`, `length`, `time`, `temperature`, `metallicity`, and `angle`. The unitless `dimensionless` dimension, which we use to represent scalars is also technically a base dimension, although a trivial one.

For each dimension, we choose a base unit. Our system's base units are grams, centimeters, seconds, Kelvin, metal mass fraction, and radian. All units can be described as combinations of these base dimensions along with a conversion factor to equivalent base units.

The choice of CGS as the base unit system is somewhat arbitrary. Most unit systems choose SI as the reference unit system. We use CGS to stay consistent with the rest of the `yt` codebase and to reflect the standard practice in astrophysics. In any case, using a *physical* coordinate system makes it possible to compare quantities and arrays produced by different datasets, possibly with different conversion factors to CGS and to code units. We go into *more detail* on this point below.

We provide sympy `Symbol` objects for the base dimensions. The dimensionality of all other units should be `sympy Expr` objects made up of the base dimension objects and the `sympy` operation objects `Mul` and `Pow`.

Let's use some common units as examples: gram (`g`), erg (`erg`), and solar mass per cubic megaparsec (`Msun / Mpc**3`). `g` is an atomic, CGS base unit, `erg` is an atomic unit in CGS, but is not a base unit, and `Msun/Mpc**3` is a combination of atomic units, which are not in CGS, and one of them even has a prefix. The dimensions of `g` are `mass` and the cgs factor is `1`. The dimensions of `erg` are `mass * length**2 * time**-2` and the cgs factor is `1`. The dimensions of `Msun/Mpc**3` are `mass / length**3` and the cgs factor is about `6.8e-41`.

We use the `UnitRegistry` class to define all valid atomic units. All unit registries contain a unit symbol lookup table (dict) containing the valid units' dimensionality and cgs conversion factor. Here is what it would look like with the above units:

```
{ "g":    (mass, 1.0),
  "erg":  (mass * length**2 * time**-2, 1.0),
  "Msun": (mass, 1.98892e+33),
  "pc":   (length, 3.08568e18), }
```

Note that we only define *atomic* units here. There should be no operations in the registry symbol strings. When we parse non-atomic units like `Msun/Mpc**3`, we use the registry to look up the symbols. The unit system in yt knows how to handle units like `Mpc` by looking up unit symbols with and without prefixes and modify the conversion factor appropriately.

We construct a `Unit` object by providing a string containing atomic unit symbols, combined with operations in Python syntax, and the registry those atomic unit symbols are defined in. We use sympy's string parsing features to create the unit expression from the user-provided string. Here's how this works on `Msun/Mpc**3`:

```
>>> from sympy.parsing.sympy_parser import parse_expr
>>> unit_expr = parse_expr("Msun/Mpc**3")
>>> from sympy.printing import print_tree
>>> print_tree(unit_expr)
    Mul: Msun/Mpc**3
    +-Symbol: Msun
    | comparable: False
    +-Pow: Mpc**(-3)
      +-Symbol: Mpc
      | comparable: False
```

(continues on next page)

(continued from previous page)

```
      +-Integer: -3
        real: True
        ...
```

When presented with a new unit specification string, a new `Unit` is created by first decomposing the unit specification into atomic unit symbols. This may require considering SI prefixes, which we allow for a whitelisted subset of atomic unit symbols, listed in the table of unit symbols below. The `Unit` instance is then created by combining a sympy expression for the unit and the appropriate CGS factors, found by combining the CGS factors of the base unit and optional SI prefixes.

`Unit` objects are associated with four instance members, a unit `Expression` object, a dimensionality `Expression` object, a `UnitRegistry` instance, and a scalar conversion factor to CGS units. These data are available for a `Unit` object by accessing the `expr`, `dimensions`, `registry`, and `cgs_value` attributes, respectively.

`Unit` provides the methods `same_dimensions_as`, which returns True if passed a `Unit` object that has equivalent dimensions, `get_cgs_equivalent`, which returns the equivalent cgs base units of the `Unit`, and the `is_code_unit` property, which is `True` if the unit is composed purely of code units and `False` otherwise. `Unit` also defines the `mul`, `div`, `pow`, and `eq` operations with other unit objects, making it easy to compose compound units algebraically.

The `UnitRegistry` class provides the `add`, `remove`, and `modify` methods which allows users to add, remove, and modify atomic unit definitions present in `UnitRegistry` objects. A dictionary lookup table is also attached to the `UnitRegistry` object, providing an interface to look up unit symbols. In general, unit registries should only be adjusted inside of a code frontend, since otherwise quantities and arrays might be created with inconsistent unit metadata. Once a unit object is created, it will not recieve updates if the original unit registry is modified.

We also provide a singleton `default_unit_registry` instance that frontend developers can copy and modify to build a simulation-specific unit symbol registry.

The `YTArray` class works by tacking a `Unit` object onto an `ndarray` instance. Besides the conversion methods already listed, most of the implementation of YTArray depends on defining all possible `ndarray` operations on YTArray instances. We want to preserve normal ndarray operations, while getting the correct units on the resulting `YTArray` (be it in-place or a copy). The proper way to handle operations on ndarray subclasses is explained in the NumPy docs page, subclassing ndarray. We follow this approach and describe the desired behavior in the next section below.

The code for these new classes will live in a new top-level `yt.units` package. This package will contain five submodules:

- `unit_lookup_table`

  Contains all static unit metadata used to generate the sympy unit system

- `unit_object`

  Contains the `Unit` class

- `unit_registry`

  Contains the `UnitRegistry` class

- `yt_array`

  Contains the `YTArray` and `YTQuantity` classes.

- `unit_symbols`

  Contains a host of predefined unit quantities, useful for applying units to raw scalar data.

11.4.4 Creating YTArray and YTQuantity instances

In the current implementation, there are two ways to create new array and quantity objects, via a constructor, and by multiplying scalar data by a unit quantity.

Class Constructor

The primary internal interface for creating new arrays and quantities is through the class constructor for YTArray. The constructor takes three arguments. The first argument is the input scalar data, which can be an integer, float, list, or array. The second argument, `input_units`, is a unit specification which must be a string or `Unit` instance. Last, users may optionally supply a *UnitRegistry* instance, which will be attached to the array. If no *UnitRegistry* is supplied, the *default_unit_registry* is used instead.

Unit specification strings must be algebraic combinations of unit symbol names, using standard Python mathematical syntax (i.e. `**` for the power function, not ^).

Here is a simple example of `YTArray` creation:

```
>>> from yt.units import yt_array, YTQuantity
>>> YTArray([1, 2, 3], 'cm')
YTArray([1, 2, 3]) cm
>>> YTQuantity(3, 'J')
3 J
```

In addition to the class constructor, we have also defined two convenience functions, `quan`, and `arr`, for quantity and array creation that are attached to the `StaticOutput` base class. These were added to syntactically simplify the creation of arrays with the *UnitRegistry* instance associated with a dataset. These functions work exactly like the `YTArray` and `YTQuantity` constructors, but pass the `UnitRegistry` instance attached to the dataset to the underlying constructor call. For example:

```
>>> from yt.mods import *
>>> ds = load("Enzo_64/DD0043/data0043")
>>> ds.arr([1, 2, 3], 'code_length').in_cgs()
YTArray([  5.55517285e+26,   1.11103457e+27,   1.66655186e+27]) cm
```

This example illustrates that the array is being created using `ds.unit_registry`, rather than the `default_unit_registry`, for which `code_length` is equivalent to `cm`.

Multiplication

New YTArray and YTQuantity instances can also be created by multiplying YTArray or YTQuantity instances by float or ndarray instances. To make it easier to create arrays using this mechanism, we have populated the `yt.units` namespace with predefined `YTQuantity` instances that correspond to common unit symbol names. For example:

```
>>> from yt.units import meter, gram, kilogram, second, joule
>>> kilogram*meter**2/second**2 == joule
True
```

```
>>> from yt.units import m, kg, s, W
>>> kg*m**2/s**3 == W
True
```

```
>>> from yt.units import kilometer
>>> three_kilometers = 3*kilometer
```

(continues on next page)

(continued from previous page)

```
>>> print three_kilometers
3.0 km
```

```
>>> from yt.units import gram, kilogram
>>> print gram+kilogram
1001.0 g
>>> print kilogram+gram
1.001 kg
>>> print kilogram/gram
1000.0 dimensionless
```

11.4.5 Handling code units

If users want to work in code units, they can now ask for data in code units, just like any other unit system. For example:

```
>>> dd["density"].in_units("code_mass/code_length**3")
```

will return the density field in code units.

Code units are tightly coupled to on-disk parameters. To handle this fact of life, the `yt` unit system can modify, add, and remove unit symbols via the `UnitRegistry`.

Associating arrays with a coordinate system

To create quantities and arrays in units defined by a simulation coordinate system, we associate a `UnitRegistry` instance with `StaticOutput` instances. This unit registry contains the metadata necessary to convert the array to CGS from some other known unit system and is available via the `unit_registry` attribute that is attached to all `StaticOutput` instances.

To avoid repetitive references to the `unit_registry`, we also define two new member functions in the `StaticOutput` base class, `quan` and `arr`. These functions simply pass the appropriate `unit_registry` object to the `YTQuantity` and `YTArray` constructors, returning the resulting quantity or array.

We have modified the definition for `set_code_units` in the `StaticOutput` base class. In this new implemenation, the predefined `code_mass`, `code_length`, `code_time`, and `code_velocity` symbols are adjusted to the appropriate values and `length_unit`, `time_unit`, `mass_unit`, `velocity_unit` attributes are attached to the `StaticOutput` instance. If there are frontend specific code units, like MHD units, they should also be defined in subclasses by extending this function.

Mixing modified unit registries

It becomes necessary to consider mixing unit registries whenever data needs to be compared between disparate datasets. The most straightforward example where this comes up is a cosmological simulation time series, where the code units evolve with time. The problem is quite general – we want to be able to compare any two datasets, even if they are unrelated.

We have designed the unit system to refer to a physical coordinate system based on CGS conversion factors. This means that operations on quantities with different unit registries will always agree since the final calculation is always performed in CGS.

The examples below illustrate the consistency of this choice:

```
>>> from yt.mods import *
>>> pf1 = load('Enzo_64/DD0002/data0002')
>>> pf2 = load('Enzo_64/DD0043/data0043')
>>> print pf1.length_unit, pf2.length_unit
128 Mpccm/h, 128 Mpccm/h
>>> print pf1.length_unit.in_cgs(), pf2.length_unit.in_cgs()
6.26145538088e+25 cm 5.55517285026e+26 cm
>>> print pf1.length_unit*pf2.length_unit
145359.100149 Mpccm**2/h**2
>>> print pf2.length_unit*pf1.length_unit
1846.7055432 Mpccm**2/h**2
```

Note that in both cases, the answer is not the seemingly trivial $128^2 = 16384\,\mathrm{Mpccm}^2/\mathrm{h}^2$. This is because the new quantity returned by the multiplication operation inherits the unit registry from the left object in binary operations. This convention is enforced for all binary operations on two YTarray objects. In any case, results are always consistent in CGS:

```
>>> print (pf1.length_unit*pf2.length_unit).in_cgs()
3.4783466935e+52 cm**2
>>> print pf1.length_unit.in_cgs()*pf2.length_unit.in_cgs()
3.4783466935e+52 cm**2
```

11.4.6 Handling cosmological units

We also want to handle comoving length units and the hubble little "h" unit. In `StaticOutput.set_units`, we implement this by checking if the simulation is cosmological, and if so adding comoving units to the dataset's unit registry. Comoving length unit symbols are still named following the pattern "(length symbol)cm", i.e. "pccm".

The little "h" symbol is treated as a base unit, `h`, which defaults to unity. `StaticOutput.set_units` should update the `h` symbol to the correct value when loading a cosmological simulation.

11.4.7 LaTeX printing

We will make use of sympy's LaTeX pretty-printing functionality to generate axis and colorbar labels automatically for unit symbols. The LaTeX strings used for atomic units are encoded in the `latex_symbol_lut`. This is necessary because, for the purposes of LaTeX representation, sympy interprets symbol names as if they were algebraic variables, and so get displayed using an italic font. Since our symbols represent units, we want to display them in a roman font and need to wrap them in `\rm{}`. New units do not need to be explicitly added to the look-up-table, by default the LaTeX symbol will simply be the string name of the unit, wrapped using `\rm{}`.

Using these LaTeX representations of atomic unit symbols, we then use sympy to generate labels, composing the LaTeX expressions for compound units according to the algebraic relationships between the atomic unit symbols.

11.4.8 YTArray operations

When working interactively, it is important to make sure quick workflows are possible. To this end, we want to make it possible to use our new dimensionful operations while still leveraging the syntactic simplicity NumPy offers. We want to avoid mandating that all user-defined data be a `YTArray` or `YTQuantity`.

To this end, we define operations between native Python objects like float, NumPy float, NumPy ndarray, and `YTArray`. In the table below, we have enumerated all combinations of `YTArray`, scalar (native Python float or np.float64), and `ndarray` for binary operations. In most cases, unitful operations are well defined, however in cases where the unitful operations are not well defined, we raise a new exception, `YTInvalidUnitOperation`.

Since NumPy defines in-place, left, and right versions of all mathematical operations (i.e. add, iadd, ladd, radd), we only list the 'basic' version of each operation, with the expectation the implemenation accounts for all four variants, which all have the same behavior with respect to passing units.

Operation	Combination	Result (pseudocode)
mul, div, truediv, floordiv	scalar, YTArray ndarray, YTArray	YTArray, units = input_units (op) 1
	YTArray, YTArray	YTArray, units = left_units (op) right_units
add, sub	scalar, YTArray ndarray, YTArray	if YTArray is dimensionless: return YTArray

	YTArray, YTArray	if left_units same dimensions as right_units:
		return YTArray, in left_units
		else:
		raise YTInvalidUnitOperation

pow	scalar, YTArray
	ndarray, YTArray
		if YTArray is dimensionless:
		return scalar**YTArray
		else:
		raise YTInvalidUnitOperation

	YTArray, scalar	return YTArray**scalar (note units change)
	YTArray, ndarray	if YTArray is dimensionless: return YTArray**ndarray raise YTInvalidUnitOperation[1]
	YTArray, YTArray	if YTArray and YTArray are dimensionless: return YTArray**YTArray raise YTInvalidUnitOperation[1]

le, lt, ge, gt, eq	scalar, YTArray ndarray, YTArray	if YTArray is dimensionless:
		return YTArray
		else
		raise YTInvalidUnitOperation

	YTArray, YTArray	if left_units same dimensions as

Now we list the behavior of unary operations on YTArray objects.

Operation	Result (pseudocode)
abs, sqrt neg	YTArray
exp	if YTArray is dimensionless: return exp(YTArray) raise YTInvalidUnitOperation

11.4.9 Unit symbol names

In the table below we provide a listing of all units that are in the current implementation. We also list the dimensions of the unit, if the unit is in the whitelist to be prefixable with SI abbreviations, the dimensions of the unit, and the adopted CGS conversion factor.

Unit	Symbol name	Dimensions	SI Prefixable?	CGS Conversion factor
Base units				
Gram	g	mass	yes	1.0
Meter	m	length	yes	100.0
Second	s	time	yes	1.0
Kelvin	K	temperature	yes	1.0
Radian	radian	angle	no	1.0
Gauss	gauss	magnetic_field	yes	1.0
Code units				
Code mass units	code_mass	mass	no	?
Code length units	code_length	length	no	?
Code time units	code_time	time	no	?
Code velocity units	code_velocity	velocity	no	?
Code magnetic field units	code_magnetic	magnetic_field	no	?
Code temperature units	code_temperatre	temperature	no	?
Code metallicity units	code_metallicity	metallicity	no	?
Normalized domain units	unitary	length	no	Domain width
Misc CGS				
Dyne	dyne	force	yes	1.0
Erg	erg	energy	yes	1.0
Electrostatic unit	esu	(energy*length)**0.5	yes	1.0
Gauss	gauss	magnetic_field	yes	1.0
Misc SI				
Joule	J	energy	yes	1.0e7
Watt	W	power	yes	1.0e7
Hertz	Hz	rate	yes	1.0

Continued on next page

[1] This one is a little tricky, since it is defined for ndarrays. Technically, it's a well-defined unitful operation if the ndarray is the exponent. Unfortunately, this will make all the elements of the ndarray have different units, so we don't allow it in practice.

Table 1 – continued from previous page

Unit	Symbol name	Dimensions	SI Prefixable?	CGS Conversion factor
Imperial units				
Foot	ft	length	no	30.48
Mile	mile	length	no	160934
Cosmological "units"				
Little h	h	dimensionless	no	?
Time units				
Minute	min	time	no	60
Hour	hr	time	no	3600
Day	day	time	no	86400
Year	yr	time	yes	31557600
Solar units				
Solar mass	Msun	mass	no	1.98841586e33
Solar radius	Rsun	length	no	6.9550e10
Solar luminosity	Lsun	power	no	3.8270e33
Solar temperature	Tsun	temperature	no	5870.0
Solar metallicity	Zsun	metallicity	no	0.02041
Astronomical distances				
Astronomical unit	AU	length	no	1.49597871e13
Light year	ly	length	no	9.4605284e17
Parsec	pc	length	yes	3.0856776e18
Angles				
Degree	degree	angle	no	$\pi/180$
Arcminute	arcmin	angle	no	$\pi/10800$
Arcsecond	arcsec	angle	no	$\pi/648000$
Milliarcsecond	mas	angle	no	$\pi/648000000$
Physical units				
Electron volt	eV	energy	no	1.602176562e-12
Atomic mass unit	amu	mass	no	1.660538921e-24
Electron mass	me	mass	no	9.10938291e-28

11.5 Testing

We have written a set of unit tests that check to make sure all valid and invalid unit operations succeed or fail as appropriate. We will also need to verify that the extant unit and answer tests pass before this can be accepted.

11.6 Backwards Compatibility

This is a serious break in backwards compatibility. Once this is accepted, units will no longer be stored in the `StaticOutput` dict. This means that all scripts which use the `pf[unit]` construction will no longer be valid. We will also need to eliminate instances of this construction within the `yt` codebase.

We will need to check to make sure the analysis modules and external tools that operate on `yt` data can either work appropriately with `YTArray` or figure out a way to degrade to `ndarray` gracefully.

CHAPTER 12

YTEP-0012: Halo Redesign

12.1 Abstract

Created: March 7, 2013

Author: Britton Smith, Cameron Hummels, Chris Moody, Mark Richardson, Yu Lu

In yt 3.0, operations relating to the analysis of halos (halo finding, merger tree creation, and individual halo analysis) will be brought together into a single framework. This will enable the analysis of individual halos through time without the need to bridge the gap between halo finding, merger tree creation, and halo profiling on one's own.

12.2 Status

Completed

12.3 Project Management Links

- Issue tracker

12.4 Detailed Description

12.4.1 Halo Analysis in yt 2.x

Currently, analyzing halos from a cosmological simulation works in the following way. First, a halo finder is run, which produces a halo catalog in the form of an ascii file. Each of the halo finders implemented in yt produce halo catalogs with slightly different formats, including various quantities that also differ. To perform any additional analysis on the halos that have been found, one then uses the `HaloProfiler`. The `HaloProfiler` reads the various halos catalogs from their files and performs a limited set of specific functionality, namely one-dimensional radial profiles

and projections. There is also a function that accepts a callback function for performing custom analysis. The analysis products for each of these are stored in separate locations. Any figures to be made from these analyses require the user to write their own scripts that are responsible for all file i/o, sorting, and plotting.

Analysis of a halo as it evolves over time first requires the creation of a merger tree. For this to work, the particles that belong to each halo need to have been written out during the halo finding step. Most of the merger trees work by manually specifying a list of dataset filenames over which the merger tree is to be calculated. A separate database file is created that contains the entire merger tree and helper functions exist to tracks a given halo's lineage over time. There is little comprehensive framework for performing halo time series analysis. With the functionality existing currently, halo time series analysis can be managed in one of two ways. The first, and more expensive by far, is to run the `HaloProfiler` on all halos in all datasets and then use the merger tree database file to correlate halos from multiple times in the user's hand-built plotting script. The second is to use the merger tree information to specify a single halo index to be analyzed with the `HaloProfiler`. This is accomplished by creating a filter for a specific halo index, and cannot account for halos coming from multiple parents or having multiple children. There are numerous other ways in which these approaches are very limiting.

12.4.2 Proposed Halo Analysis in yt 3.0

All of the functionality described above will be managed by a series of new objects working in a hierarchy. These will be `HaloCatalogTimeSeries`, `HaloCatalog`, and `Halo`, decribed in further detail below. The files created by the operations described here will allow for the full state of the `HaloCatalogTimeSeries` object to be restored by running the same commands that were used to create them. This will allow the user to create a single script that will be run first on a supercomputer to perform all of the dataset-requiring analysis and then later on a local machine to load the halo data into memory for further analysis and plotting.

`HaloCatalogTimeSeries`

This object will accept a `TimeSeriesData` object containing all the datasets to be used. Its two primary functions will be to perform halo finding on all datasets and creating the merger tree. Each of these two steps will be performed with separate functions calls where the arguments given will be a callable halo finding or merger tree function and a dictionary containing additional configuration paramters specific to the provided callables. The data structure contained in memory will be a time-sorted list of `HaloCatalog` objects, one for each dataset. It will also contain a dictionary of dictionaries showing the merger tree information for each halo. The on-disk format for `HaloCatalogTimeSeries` objects will likely need to be refined, but will ideally preserve the system of pointers connecting `Halo` objects to their past and future counterparts (described further in the `Halo` section). The data stored here will potentially be far too large for a single file. Instead, a system of multiple files that is capable of quickly reconstructing the `Halo` pointer structure may be better.

The other primary function will be to create halo tables that are flattened numpy structured arrays of various halo quantities from all or a selection of all halos (e.g. by timestep) from all halo catalogs. This will enable easy slicing and plotting of properties from multiple halos.

`HaloCatalog`

This will be a light-weight container for `Halo` objects from a single dataset. It will be responsible for writing `Halo` objects to and restoring from disk. It will be a numpy structured array. The manner in which `HaloCatalog` objects will be written to disk will be specified similar to how halo finders and merger tree generators given to `HaloCatalogTimeSeries` objects, i.e., by providing a callable writer function. This will allow users to write to any number of standardized formats, such as the IRATE format.

Halo

The Halo object will contain all quantities associated with a given halo calculated either during the halo finding step or by analysis performed later. By default, particle information will be saved to disk after halo finding has been performed since it is required for merger tree generation. However, particle information will not remain attached to the Halo object since it takes a great deal of memory to store this information. Instead, there will be several halo quantities calculatd at instantiation using these particles including center-of-mass phase-space coordinates, densest point, shape of halo, and merger tree information (matching against previous and later timesteps to determine lineage of a halo). However, the option will exist for reloading particle information for later analysis. This technique of frontloading analysis that requires particle information is how some halo finders, such as Rockstar, currently operate.

The Halo object will also have pointers to the Halo objects that are their past and future instances, essentially the most massive pro/postgenitors with the largest match of particles inventories. This will allow one to perform any additional analysis on a single halo lineage simply by traversing this linked list. Further analysis on Halo objects will be facilitated by associating quantities with callable functions. If the user attempts to access a halo quantity whose value is not currently stored within the Halo object, it will run the associated callable to create it. At any time, the HaloCatalogTimeSeries object can be directed to update the files on disk with any new quantities that have been calculated.

12.5 Backwards Compatibility

This will be a completely new framework for performing this type of analysis. Other than working with existing halo finders and potentially reading in the output they produce now, there will be no compatibility with pre-existing machinery. For these reasons, this development will be confined to yt 3.0.

12.6 Current Progress

Development of the new halo analysis is taking place in this repository under the **ytep0012** bookmark. This work is being done alongside the unit refactor and thus includes all changes in the **unitrefactor** bookmark here. The majority of the work is taking place within yt/analysis_modules/halo_analysis. Everything detailed below, except where explicitly noted, has been implemented.

12.6.1 HaloCatalogTimeSeries

Not Implemented. This is currently awaiting development of a new merger tree framework.

12.6.2 HaloCatalog

This relies on the recently added ability to load a Rockstar halo catalog as a yt dataset, referred to hereon as a halo finder dataset for clarity. A HaloCatalog object is created by providing it with a simulation dataset, a halo finder dataset, or both.

```
dpf = load("DD0064/DD0064")
hpf = load("rockstar_halos/halos_64.0.bin")

hc = HaloCatalog(halos_pf=hpf, data_pf=dpf,
                 output_dir="halo_catalogs/catalog_0064")
```

If the halo_pf is not given, halo finding will be done using the method provided with the `finder_method` keyword (**not implemented**). A data container can also be given, associated with either dataset, to control the spatial region in which halo analysis will be performed.

Analysis is done by adding actions to the `HaloCatalog`. Each action is represented by a callback function that will be run on each halo. There are three types of actions.

1. **Quantities** - a call back that returns a value or values. The return values are stored within the halo object in a dictionary called "quantities." At the end of the analysis, all of these quantities will be written to disk as the final form of the generated "halo catalog."

```
# definition of the center of mass quantity
def center_of_mass(halo):
    if halo.particles is None:
        raise RuntimeError("Center of mass requires halo to have particle data.")
    return (halo.particles['particle_mass'] *
            np.array([halo.particles['particle_position_x'],
                      halo.particles['particle_position_y'],
                      halo.particles['particle_position_z']])).sum(axis=1) / \
                               halo.particles['particle_mass'].sum()
# add to a registry of available quantities
add_quantity('center_of_mass', center_of_mass)

# in the actual halo analysis script
hc.add_quantity("center_of_mass")
```

The above example is better suited as a parallel-safe derived quantity, but the use is the same.

Instead of being generated from a callback, quantities can also be values pulled directory from the halo finder dataset.

```
# get the field value ("halos", "particle_mass") for this halo from the halo dataset
hc.add_quantity("particle_mass", field_type="halos")
```

2. **Callbacks** - the callback is actually the super class for quantities and filters and is a general purpose function that does something, anything, to a `Halo` object. This can include hanging new attributes off the `Halo` object, performing analysis and writing to disk, etc. A callback does not return anything. In the example below, we create a sphere for a halo with a radius that is twice the saved "virial_radius" (in the quantities dict), recenter it on the location of maximum density, then do some profiling, compute virial quantities based on those profiles (storing them in the quantities dict), and then write the profiles to disk.

```
hc.add_callback("sphere", radius_field="virial_radius", factor=2.0)
hc.add_callback("sphere_field_max_recenter", ("gas", "density"))
hc.add_callback("profile", "radius", [("gas", "matter_mass"),
                                      ("index", "cell_volume")],
                weight_field=None, accumulation=True,
                output_dir="profiles", storage="profiles")
hc.add_callback("virial_quantities", ["radius", ("gas", "matter_mass")])
hc.add_callback("save_profiles", storage="profiles")
```

Currently existing stock callbacks:

- sphere creation
- sphere recenter
- sphere bulk velocity
- 1D profiling
- virial quantity calculation based on 1D profiles

- writing profile data
- reloading saved profile data
- removing `Halo` attributes.
- PhasePlot

3. **Filters** - a filter is a callback function that returns True or False. If the return value is True, any further queued analysis will proceed and the halo in question will be added to the final catalog. If the return value False, further analysis will not be performed and the halo will not be included in the final catalog.

```
hc.add_filter("quantity_value", "matter_mass_200", ">", 1e13, "Msun")
```

Currently existing stock filters:

- quantity filter (shown above): uses an eval statement with a value stored in the quantities dict

Running and Order of Operations

After all callbacks, quantities, and filters have been added, the analysis begins with a call to `HaloCatalog.create`.

```
hc.create(save_halos=False, njobs=-1, dynamic=True)
```

The `save_halos` keyword determines whether the actual `Halo` objects are saved after analysis on them has completed or whether just the contents of their quantities dicts will be retained for creating the final catalog. The looping over halos uses a call to `parallel_objects` allowing the user to control how many processors work on each halo. The final catalog is written to disk int the output directory given when the `HaloCatalog` object was created. The final halo catalog can then be loaded in as a yt dataset just in the manner of a halo finder dataset.

All callbacks, quantities, and filters are stored in an "actions" list, meaning that they are executed in the same order in which they were added. This enables the use of simple, reusable, single action callbacks that depend on each other. This also prevents unecessary computation by allowing the user to add filters at multiple stages to skip remaining analysis if it is not warranted.

Reloading a Halo Catalog

A `HaloCatalog` saved to disk can be reloaded as yt dataset with the standard call to `load`. Any side data, such as profiles, can be reloaded with a `load_profiles` callback and a call to `HaloCatalog.load`.

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

hpf = load("halo_catalogs/catalog_0046/catalog_0046.0.h5")
hc = HaloCatalog(halos_pf=hpf,
                 output_dir="halo_catalogs/catalog_0046")
hc.add_callback("load_profiles", output_dir="profiles",
                filename="virial_profiles")
hc.load()
```

The `load` and `create` functions are wrappers around a `_run` function responsible for looping over all the halos and applying callbacks. The only difference between the two is that their default keyword arguments are specifically tailored for creating (do not retain `Halo` objects, do write catalog) or rereading catalogs (do retain `Halo` objects, do not write catalog).

12.6.3 `Halo`

`Halo` objects are created by the `HaloCatalog` during the call to `HaloCatalog.run`. They are mainly meant to be temporary objects for retaining information so that it can be passed between callbacks. They can be kept by specifying *save_halos*=*True*. This might be useful when working with a time series of halo catalogs where *future_self* and *past_self* attributes may act as pointers to `Halo` objects within `HaloCatalogs` that are time-adjacent.

12.7 Remaining Work

See the Trello board.

All are welcome to get involved with development, testing, etc!

CHAPTER 13

YTEP-0013: Deposited Particle Fields

13.1 Abstract

Created: April 25, 2013

Author: Chris Moody, Matthew Turk, Britton Smith, Doug Rudd, Sam Leitner

The majority of the yt codebase is currently built around Eulerian, grid or cell-like quantities. In order to use particle quantities, we typically have to deposit particles and essentially make them look like fluid quantities. This YTEP details the suggest deposition process, how to implement it, how to extend and subclass it, and suggested syntax.

This should improve particle support for Octrees and SPH codes dramatically, and extend particle deposition syntax for grid-patch codes.

Note that while this covers initial implementation of particle deposition, that deposition does *not* include smoothing kernels that utilize extended regions outside of the target region. SPH smoothing kernel implementations will be defiend in a subsequent YTEP.

Furthermore, we note that this describes fundamentally an interface between the particles and an *index*. For eulerian codes, the index corresponds to the fluids; however, for SPH and N-body systems, this is not the case.

13.2 Status

Completed

13.3 Project Management Links

Pull request this was conducted inside:

- https://bitbucket.org/yt_analysis/yt-3.0/pull-request/32/implement-initial-spatial-chunking-for/diff

13.4 Detailed Description

13.4.1 Particle Deposition in yt 2.x

Currently, particle deposition for grid-patch codes works by querying particle fields and supplying them to a routine like CIC_Deposit3. It is non-trivial to extend this CIC function to octree codes but essential to making SPH codes interoperable with the yt codebase.

13.4.2 Proposed Syntax

The names of deposited fields can be user-defined, and thus are not explicitly restricted. However, as having a deposited fields becomes more common in the yt framework and libraries begin to expect and depend on particular names, we suggest that field names are written as `("deposit", "pname_poperation")` where `pname` is the name of the particle type and `poperation` is some semantically-meaningful description of the operation. `deposit` is defined as a fluid type in all frontends. This indicates that the returned array is shaped like a fluid field and not particle-shaped. This is distinct from `gas` as we may have conflicting or overlapping field definitions.

13.4.3 Example Deposited Field

Below is example particle deposition field defined in Python:

```
@derived_field(name = ("deposit", "particle_count"),
               validators=[ValidateSpatial()])
def particle_count(field, data):
    pos = np.column_stack([data["particle_position_%s" % ax]
                           for ax in 'xyz'])
    return data.deposit(pos, method = "count")
```

13.4.4 Changes to Frontend Code

We exploit the fact that the octree frontends share a common base class `OctreeSubset(YTSelectionContainer)` to create a common function `deposit()`. The patch-based codes have an analogous `AMRGridPatch(YTSelectionContainer)`. The deposition is passed the particle positions, the particle fields required, and the deposition method: `deposit(positions, field = None, method = None)`. The deposit function uses `method` to lookup a Cython `ParticleDepositOperation` class in `particle_deposit.pyx`. This class defines the deposition procedure in three steps, which `deposit` calls sequentially. The first `ParticleDepositOperation` member function is `initialize` which allocates the memory required to hold temporary arrays for the deposition of particles into grids or octs. Extra temporary arrays are useful when a reduction of data must occur after the we have looped through all particles. The next step is either `process_octree` or `process_grid` where we loop over all particles, find the oct or cell in an octree or grid (respectively). Once found, we call `process(dims, left_edge, dds, 0, pos, field_vals)` which relates a single particle, its associated cell, and the incremental deposited value. The last step `finalize` reduces the data from the temporary arrays and return an oct-shaped or grid-shaped array. This organization allows us to seperate the particle lookup along in a grid or oct tree from the deposition operation we would like to perform.

13.4.5 Example Cython Code

Below we include an example of the base particle deposit class with most of the Cython type definitions removed for legibility:

```
cdef class ParticleDepositOperation:
    def initialize(self, *args): raise NotImplementedError
    def finalize(self, *args): raise NotImplementedError
    def process_octree(self, octree, dom_ind, positions, fields = None):
        for i in range(positions.shape[0]):
            oct = octree.get(pos, &oi)
            offset = dom_ind[oct.ind]
            self.process(dims, oi.left_edge, oi.dds,
                         offset, pos, field_vals)
    def process_grid(self, gobj, positions,fields = None):
        for i in range(positions.shape[0]):
            for j in range(3):
                pos[j] = positions[i, j]
            self.process(dims, left_edge, dds, 0, pos, field_vals)
    def process(self, *args): raise NotImplementedError
```

Below we subclass the template above to deposit a particle count, taking care to override `initialize`, `process` and `finalize` but leaving grid traversal in `process_octree/grid` alone, ensuring that this will work with grid and octree codes:

```
cdef class CountParticles(ParticleDepositOperation):
    def initialize(self):
        self.ocount = np.zeros(self.nvals, dtype="float64")
        cdef np.ndarray arr = self.ocount
        self.count = <np.float64_t*> arr.data
    @cython.cdivision(True)
    cdef void process(self, int dim[3],left_edge[3], dds[3], offset,
                      ppos[3], *fields):
        cdef int ii[3], i
        for i in range(3):
            ii[i] = <int>((ppos[i] - left_edge[i])/dds[i])
        self.count[gind(ii[0], ii[1], ii[2], dim) + offset] += 1
    def finalize(self):
        return self.ocount
```

Using the templates and organizational scheme proposed here, one can define fields with arbitrary particle selections (e.g. young stars), perform arbitrary accumulations (e.g. count, sum, or std), loops over all of the particles multiple times, and switch between cloud-in-cell, SPH smoothing kernel, or simple direct deposition.

13.4.6 Future SPH Kernel

A process very similar to this will be utilized in the future to conduct smoothing kernel operations. This will require two operations:

- Iteration over the Octs, rather than the particles, and selection of particles based on proximity to an Oct
- An octree selector that has lee-way in its selection of particles; i.e., particles can be fed in as having a `dx` that allows them to be selected by octs within which they do not directly reside.

We may find that this specific operation is too slow for applying the smoothing kernel, in which case other options will be explored.

An initial implementation of this operation is contained in `yt/geometry/particle_smooth.pyx`.

13.5 Backwards Compatibility

This has no backwards incompatible changes.

13.6 Alternatives

We were unable to identify any.

CHAPTER 14

YTEP-0014: Field Filters

14.1 Abstract

Created: July 2nd, 2013 Author: Matthew Turk

This YTEP outlines a method for defining generic, evaluated filters to apply to particles used in derived fields. Currently it does not extend to fluid quantities, as that will require rethinking the method of presenting and handling Eulerian quantities.

14.2 Status

Proposed. Target is 3.0a3.

14.3 Project Management Links

There has been some disucssion of this in YTEP-0013 and its pull request.

- https://bitbucket.org/yt_analysis/ytep/pull-request/15/ytep-0013-first-class-deposited-particle/diff
- https://ytep.readthedocs.org/en/latest/YTEPs/YTEP-0013.html
- https://bitbucket.org/yt_analysis/yt-3.0/pull-request/59/

14.4 Detailed Description

Currently, filtering particles is done ad-hoc by derived fields. Typically something like this is done:

```
def finest_particles(field, data):
    filter = data["ParticleMassMsun"] <= 340000
    pos = data["Coordinates"][filter, :]
    d = data.deposit(pos, [data["all", "ParticleMass"][filter]],
                     method = _method)
    d /= data["CellVolume"]
    return d
```

This is not ideal, as it requires new fields to be defined for every single particle filtering *and* field combination. This requires every single derived field that is desired to be filtered individually, including derived fields that are used as dependencies in another field. This is not workable, and a new mechanism for defininig filtered particle types is needed. However, rather than declaring a completely new domain-specific language for defining particles to select inside a given field specification, this YTEP defines a method for declaring filters that can be applied to particles inside a contextmanager. This means that all particles used *inside* the context manager will be pre-filtered.

However, to avoid over-complication, the filtering step will be defined inside functions similar to derived fields and will *not* be auto-detected. Instead, all filters defined will be allowed to be applied and in the case of a filtering-needed field not being found, an exception will be allowed to be raised. However, field dependencies noted in the creation of a filter will be taken into account when filters are added to a given dataset.

These filters will be added and viewed as a new particle type. For instance, if a dataset has only "all" particles, a new filter could be added that filtered out particles that should be regarded as "star" particles and that filter will then be presented as a new particle type "star".

Filters are only meant to act on homogeneous groups of particles. For instance, a given filter could not select sets of particles with hetereogeneous attributes and combine them.

Here is an example filter that would accomplish the filtering task shown above:

```
def finest_particles(pfilter, data):
    filter = data["all","ParticleMassMsun"] <= 34000
    return filter

add_particle_filter("finest_particles",
     function = finest_particles,
     filtered_type = "all",
     requires=["ParticleMassMsun"])

ds = load("DD0040/DD0040")
sp = ds.h.sphere("max", (1.0, 'mpc'))
sp.quantities["TotalQuantity"]( [("finest_particles", "ParticleMassMsun")] )
```

However, more complex filters could be defined as well, relying on additional fields. Furthermore, a side-effect of future particle field definitions being generated for specific particle types (described in issue 598) would be that any particle filters defined would also have any particle derived fields added on a per-particle-type basis added automatically. The addition of field definitions will occur during the creation of derived fields.

Note that we do explicitly specify field dependences in these particle types. This may cause issues, as derived fields will first need to be identified, then particle filters, then any derived fields for those new ad-hoc particle types will be added. This will require some refactoring of field detection methods, which will overall serve to improve the reliability of the code base and field detection mechanisms.

Derived fields based on filtered particles are not currently available; only derived fields that work on the filtered particle type will be available.

Adding this filtering mechanism will also considerably simplify the Enzo frontend, as currently the Enzo frontend defines several different methods for identify star particles. (Other frontends, where attributes of particles separate them into different classes, will also benefit.) As an example, for Enzo 2.X-class simulations, the definition of a star is

through *either* a `creation_time` attribute or a `particle_type` attribute. This will enable definition of filters, and only the one that is applicable to the specific dataset will be added and applied.

```
def star_creation_time(pfilter, data):
    filter = data["all", "creation_time"] > 0
    return filter
add_particle_filter("star",
     function = star_creation_time,
     filtered_type = "all",
     requires = ["creation_time"])

def star_particle_type(pfilter, data):
    filter = data["all", "particle_type"] == 2
    return filter
add_particle_filter("star",
     function = star_particle_type,
     filtered_type = "all",
     requires = ["particle_type"])
```

The correct filter will be identified and added to a dataset. Filters are distinct from types in the sense that types have a fast-path that can be passed down to IO functions; for instance, this may be because particles are stored in a separate location or IO routines are able to quickly identify those particles that are able to be loaded. Filters are better thought of as a set of generic validation and selection routines, where those filters are difficult or impossible to pass into the IO routines in a general way.

Since this is a multi-map to filter names, we will not be able to store filters in a dict-like object, or we will at the very least have to return a list of possible filters when accessing via dict. This will likely not serve as a large barrier, as the set of filters will not be user-exposed.

In addition to this, we will define a similar system for filters as is done for fields, in that a hierarchy of filtering databases will be available. The base or universal filters will be available across codes (suitable, for instance, in direct cross-code comparison) and then frontend-specific filters can be created. This will enable degeneracies of field names and so on to be eliminated. The first implementation will require manual calling of `add_particle_filter` on `StaticOutput` subclasses *before* instantiation of a hierarchy.

However, unlike derived fields, because these filters define actual new particle types, they will not by-default be applied universally, but instead universal filters will need to be activated by the user. Frontends can decide on a frontend-by-frontend basis whether or not new frontend-specific filters will be added by default.

14.5 Backwards Compatibility

This should not break any backwards compatibility by itself. However, should functions in yt begin to rely on these filters, those functions will no longer be backwards compatible.

14.6 Alternatives

I have not presently identified any alternatives, other than construction of a domain-specific language for describing filters that would then be embedded in the particle type. I believe that will raise complexity considerably.

CHAPTER 15

YTEP-0015: Transfer Function Refactor

15.1 Abstract

Created: August 13, 2013 Author: Sam Skillman

This YTEP proposes a fundamental change in the way transfer functions are constructed, modified, and implemented. The overall goal is to decrease the overhead and difficulty in constructing high-quality transfer functions and displaying their current state to the user.

15.2 Status

Status should be one of the following:

1. Proposed
2. In Progress

YTEPs do not need to pass through every stage.

15.3 Project Management Links

PR Under Development: https://bitbucket.org/yt_analysis/yt/pull-request/538/transfer-function-helper/diff

15.4 Detailed Description

Transfer functions are currently:

- Fragile – to log/linear, ranges, field swapping
- Complex – must have prior knowledge of data to construct valid TF

- Difficult to Design – User must guess where interesting features will be.

The aim of this refactoring is to alleviate these three problems. To do so, we will implement several helper functions that are automate many of the actions that are commonly used during the design process of the transfer function. Several operations may be costly, and thus will not be done automatically but rather upon request by the user.

This splits up the TransferFunction into two pieces – the TransferFunction and TransferFunctionData. The former encompasses all the user-facing API in terms of designing and modifying a transfer function, and the latter contains the data needed by the volume renderer.

Suggested TF Structure:

```
class TransferFunctionData(object):
    """
    Contains the data used by the Camera to actually do the volume
    rendering.  Not accessed by the user in most circumstances.  This
    contains most of what the TransferFunction used to be.
    """

class TransferFunction(object):
    def __init__(self, data_source=None):
        self.data_source = data_source
        self.pf = self.data_source.pf
        self.rgb_field = None
        self.bounds = None
        self.alpha_field = None
        self._valid = False
        self.transfer_function_data = None

    def smart_build(self):
        """
        Automatically set up best guess bounds, and run initial 1D
            profiling of given field. We could make this as automatic
            or not as we want.
        """
        pass

    def set_field(self, field):
        """
        Sets the rgb channel to be linked to a given field, invalidating
        the current profiles/ranges if different than current field.
        """
        if field == self.rgb_field:
            return
        self._valid = False
        assert (field in self.pf.h.field_list)
        self.rgb_field = field

    def _get_field_bounds(self):
        return self.data_source.quantities['Extrema'](self.field)[0]

    def set_bounds(self, bounds=None):
        if bounds is None:
            bounds = self._get_field_bounds()
        # Do error checking on log/linear state of rendering.
        self.bounds = bounds

    def _get_1D_field_histogram(self):
        """
```

(continues on next page)

(continued from previous page)

```
        Calculates 1D profile (in mass/volume/count) of current field to
        aid in placement of transfer function features.
        """
        pass

    def plot/show/display(self):
        """
        plots, shows, or displays current TF based on how the user is
        interacting with yt.  This could save an image to tf_tmp.png,
        display in an interactive matplotlib backend, display in an IPython
        notebook, or directly interact with the user's visual cortex.
        """

    # add in all the transfer function modifiers here (gaussians, layers,
    # ramps, map_to_colormap, etc.)

    def set_log(self, log=True):
        self.log = log

    def clear(self):
        """Clears out the channel values, but leaves the bounds intact"""
        pass

    def _get_tf_data(self):
        """
        This is what the Cameras call to get the TF information.  This does
        error checking to make sure the transfer function is valid."""
        if not self._valid:
            # Rebuild TransferFunctionData
            pass
        return self.transfer_function_data
```

After this is implemented, the usage pattern I would see would be something like:

```
tf = TransferFunction(pf.h.all_data())
tf.set_field('Density')
tf.smart_build() #<--- maybe another name like: auto_build or auto_awesome
tf.display()  #<--- Should we make this automatically display if possible?

cam.set_transfer_function(tf) #<---- links a camera to this tf.
# Alternatively we could have done tf = cam.transfer_function and modified
# the camera's tf directly.'

tf.set_log(True) # <--- invalidates the TF
tf.do_whatever_modifications(...)
cam.snapshot()
```

15.5 Backwards Compatibility

This change will break backwards compatibility with how old TransferFunctions were constructed.

CHAPTER 16

YTEP-0016: Volume Traversal

16.1 Abstract

Created: September 10, 2013 Author: Matthew Turk

yt should consider volume traversal, accumulation of data, and flexible definitions of paths to be first-class operations as well as implementable by individuals. Essentially, we need a method for describing "derived values for volumes".

16.2 Status

In progress

16.3 Project Management Links

This is being done in the bookmark `cylindrical_rendering` in http://bitbucket.org/MatthewTurk/yt-3.0 .

16.4 Detailed Description

Currently, the only mechanisms for studying or understanding data in yt are contained in the following procedures:

- Derived quantities (generating scalars from fields in volumes)
- Derived fields (generating fields from other fields)
- Contour identification
- Ray casting (on or off axis; i.e., projections or volume rendering)
- Streamlines
- Contour identification

- Surface extraction

Several of these items utilize brick decomposition, but not all. This is the process by which overlapping grids are broken apart until a full tesselation of the domain (or data source) is created with no overlaps. This is done by the kD-tree decomposition.

What this YTEP proposes is to make handling tiles of data first class, as well as provide easy mechanisms for creating volume traversal mechanisms. There are two components to this: handling tiles of data, and creating fast methods for passing through the data and moving between tiles.

Note that this YTEP does not (yet) address situations where the mesh of the simulation is too large to fit into memory on a single node. Because the kD-tree is able to build in parallel, this essentially will amount to distributing regions of the dataset (where the mesh may not be known) to individual processors, allowing build on those processors, and enabling individuals to describe reduction steps in their operators.

16.4.1 Brick Iteration

Currently, in yt-3.0, all data objects expose a "block" iterator that returns data containers as well as masks of data. A similar iterator should exist for iterating over the "tiles" that compose a given data object. How this should behave is somewhat open for discussion, as the kD-tree itself has a notion of a 'viewpoint traversal' which may be important. Furthermore, it is not necessarily true that the traversal will be easily defined. As an example of this, tiles may need to traversed according to extrema in some fluid.

Traversal Orders

The order that tiles are returned to the individual should be flexible and extensible. A few predefined orders should be implemented:

- Depth-first traversal
- View-point traversal
- Breadth-first traversal

Additionally, these should allow for front-to-back or back-to-front yielding. An example API for this would be:

```
data_source = ds.h.sphere(c, (10.0, 'mpc'))
for brick in data_source.tiles.depth_traverse():
    operation(brick)
```

By default, `depth_traverse()` would also be exposed for simply iterating over the `tiles` object. Additional traversals could be extensibly defined. Many of these traversal *already* exist for the AMR kD-tree. A traversal should be able to be defined and executed from the following set of information:

- kD-tree object
- Arguments defining the traversal itself
- Data-centering (optional) and fields (optional)

It should yield `PartitionedGrid` objects.

Return Values

Most importantly, the notion of what is returned by this system needs to be defined. The notions of what brick is and possessed need to be defined. There are several options:

1. An empty `CoveringGrid` that knows how to read data.

2. A filled (i.e., data pre-specified) `PartitionedGrid`, where vertex or cell-centered data must be specified.
3. A slice object and a grid object
4. A new object, designed for this system, which acts as a superset of `PartitionedGrid`. This object would include connectivity information as well, as it would not be independent of the tree itself. The `PartitionedGrid` could be modified to fit this.

Regardless of which object is returned, at a minimum a kD-tree (or other partitioning) must be created when requested, at the call to tiles, potentially cached, and then objects iterated over. Each of these tiles is guaranteed to be the finest data available within the region they cover, and they are guaranteed not to overlap with any others.

For the purposes of this YTEP, we will assume the fourth option. If the `PartitionedGrid` object were to be extended, I believe it would likely be best to extend it as follows. Note that for many of these operations we implicitly assume that it is operating on a grid patch; for octree codes, the creation of this object will be considerably simpler, and for particle codes we simply define these as the leaf nodes from the octree index itself. Because we need to handle particle codes, we must also ensure that these objects can query particles.

- Cache a slice of the grid or data object that it operates on. (For situations where it fully encompasses the parent region, it need not have a slice.)
- Create a mechanism for filtering particles from the data object it operates on.
- Enable the object to query new fields from its source object. This means that at instantiation time we may not regard the object as having a given field, but that this field can be added at a later time by querying.
- Provide a mechanism for identifying neighbor objects from a given face index. This is the connectivity relationship described above; given any one cell that resides on the boundary of a brick, return the brick (which may or may not be a leaf node) that is adjacent. This would enable identifying the leaf node at a given location within that boundary cell, which may reside at a higher level of refinement and could thus correspond to multiple tiles. This degeneracy results from the fact that we cannot guarantee that neighboring tiles differ by only a single level of refinement. However, because this will be defined at the Python level, rather than specifically for well-defined traversal operations, this is acceptable as we should leave open to the individual how to select the appropriate cell or what to do with it.
- Provide mechanisms for generating vertex-centered data or cell-centered data quickly.

At the present time, a simple first-pass at implementation could occur with the following:

- Implement a `tiles` routine that mandates supplying fields to cache or load, the vertex or cell centering of data, and a viewpoint traversal scheme.
- Cache a kD-tree based on these tiles.
- Iteratively yield tiles from this tree based on the traversal specified above.

The interface for these tiles, at a minimum, must expose that of the `PartitionedGrid` with one modification: fields should be accessible by `__getitem__`, so that any possible changes in the future that would expose this would be backwards compatible with usages now.

16.4.2 Volume Traversal

The second aspect of this YTEP is to define a mechanism for integrating paths through tiles. Currently we do this through strict vectors that cannot be re-entrant into a grid; these vectors cannot change path along the way, and the number of them is fixed at the time of the first grid traversal.

As currently implemented, flexibility in volume traversal is defined in terms of the mechanism by which values are accumulated. This includes the definition of these objects, all inside `grid_traversal.pyx`:

- `ImageContainer`
- `ImageAccumulator`

- `sampler_function`
- Accumulator data (i.e., `VolumeRenderAccumulator`

Essentially, for a given image type, a sampler can be defined. This sample receives the following arguments: `VolumeContainer` (a C-interface to a partitioned grid with `nogil`), `v_pos` (vector position), `v_dir` (vector direction), `enter_t` (cell-entrance in terms of the parameter), `exit_t` (exit time based on the vector at time of entrance), `index` (index into the data) and `data`, a pointer to an accumulator (i.e., `ImageAccumulator`) object.

These sampler functions are called for every index that a vector traverses. The volumes themselves are traversed inside `walk_volume` (and, in the nascent cylindrical volume rendering bookmark, `walk_cylindrical_volume`). This assumes cartesian coordinates and simply calls the `sampler_function` for every zone that is crossed. This enables volume rendering, projecting and so on to be conducted simply by swapping out the sampler function and correctly interpreting an image object returned.

However, this is not sufficient for arbitrary traversals *or* arbitrary data collection. We need flexibility to define the following things:

- The mechanism of traversing blocks of data (covered at a higher level by the kD-tree itself, and not necessarily a part of this YTEP)
- Bootstrapping traversal of a volume by a given ray object. This would include identifying the zones that a ray first encounters and setting its initial time of intersection.
- Defining a mechanism for updating the indices in the volume that a ray will intersect next
- Defining a method for determining when a ray has left an object
- Defining a method for selecting the next brick to traverse or connect to
- Updating the value of a ray's direction

Many of the problems can be seen simply by considering cylindrical volume rendering itself. If the view point is somewhere outside the cylinder looking toward it, rays from an orthonormal image plane will each construct a chord through the cylindrical shells. These chords will each span up to pi along the theta direction, and can have the following properties in their traversal:

- dtheta/dl can switch signs
- grids can be periodic with themselves
- dr/dl can switch signs
- a ray can exit a grid off the r boundary and then re-enter it *later* in the computation
- Both dtheta/dl and dr/dl change with each update of the ray's position, and are not even constant over a single zone.

While this demonstrates some of the complexity, we also want to be able to support translating streamlines, clump finding and even gravitational lensing into this new mechanism for traversing volumes.

Therefore, we need a new mechanism that abstracts (independently) both the collecting or accumulating of data as well as the mechanism by which a given ray traverses a patch of data, whether that patch is one or several cells large. In this manner we will remain neutral to the nature of the data container, which may be an octree, a kD-tree, or a single grid.

Flow Control

1. At the outermost level, tiles will be traversed in Python, and a collection of rays (either in an `ImagePlane` or some other object) will be handed each brick as it comes.
2. Each ray will be "bootstrapped" onto a brick. This will result either in a traversal or an immediate return. (At a later time we will consider fast evaluation of which rays to consider.)

3. Each cell traversed by the ray will be "sampled" in some way.
4. The ray state (location, index, direction, etc) will be updated.
5. Rays will traverse until they leave a brick.
6. The next brick will be identified, either from ray positions or from the traversal at the python level.

Note that this does not *yet* enable a ray to request the next brick at a given position, which will be necessary. However, for the purposes of this iteration of the YTEP, we take it as given that such communication will be defined at a later time, or will be handled on a ray-by-ray basis, where the iteration is managed for each ray individually.

Objects to Manage

To accommodate the flow control outlined above, the following classes will need to be implemented, with the following specifications. These will be in Cython. A base class (listed below) will form the basis for each type of traversal.

```
struct ray_state:
    np.float64_t v_dir[3]
    np.float64_t v_pos[3]
    np.float64_t tmax[3]
    int ind[3]
    int step[3]
    np.float64_t enter_t
    np.float64_t exit_t
    void *sdata

class GeometryTraversal:

    # set values like domain size or whatever is necessary here
    def __init__(self, parameter_file)
    # Return whether the ray hits the vc or not
    cdef int initialize_ray(self, ray_state *ray, VolumeContainer *vc) nogil
    cdef int increment_ray(self, ray_state *ray, VolumeContainer *vc) nogil
    cdef np.float64_t intersection(self,
            np.float64_t val, int axis, np.float64_t v_dir[3],
            np.float64_t v_pos[3]) nogil
    cdef int walk_volume(self, VolumeContainer *vc, sampler_function *sampler,
                         ray_state *ray, np.float64_t *return_t = ?) nogil

 cdef
```

The `ray_state` object will be independent of the geometry, and will *always* refer to the cartesian state of the ray. A given geometry traversal will set up the ray state (i.e., where it intersects with a volume container) and how to increment the ray state as zones are crossed. The `initialize_ray` function will determine the state of the ray as it first touches a brick, and will return 0 or 1 if the ray is inside that brick. The `increment_ray` function will receive a ray and determine the crossing time in the parameter `t` that the ray uses as it passes through a cell. The return value is 0 for the ray having left the object and 1 for the ray being within the object and the sampler function needing to be called. `intersection` will get the position at which a ray intersects a given value, and `walk_volume` will typically be described in the base class and not overridden elsewhere. Part of the level of abstraction is to enable `walk_volume` to largely be the same for each geometry, but enabling it to be overridden means we can use the same traversal for other operations such as clump finding and so on.

As a first implementation, the following classes will need to be implemented:

- `CartesianTraversal`
- `PolarTraversal`

- `CylindricalTraversal`

At a later time, the `SphericalTraversal` object can be implemented.

16.5 Backwards Compatibility

This should retain all backwards compatibility for cartesian systems.

16.6 Alternatives

I'm not sure of any alternatives currently.

CHAPTER 17

YTEP-0017: Domain-Specific Output Types

17.1 Abstract

Created: September 18, 2013 Author: Matthew Turk and Anthony Scopatz

This YTEP is designed to begin the process of generalizing astrophysics-specific components of yt toward applications in other domains.

17.2 Status

Proposed and in completed.

This would only be implemented in yt 3.0.

17.3 Project Management Links

The first phase pull request, which is contingent on this being accepted, is here:

- https://bitbucket.org/yt_analysis/yt-3.0/pull-request/96/rename-generic-objects/diff

17.4 Detailed Description

Currently, yt is extremely strongly focused on astrophysical data. This leads to the inclusion of attributes such as `cosmological_simulation`, `current_redshift` and so on, as well as some other fundam. Even within astrophysical simulations, these can be irrelevant or unnecessary. Furthermore, there may be attributes relevant to other domains (that transcend a single subclass of `StaticOutput`) that may be relevant or necessary.

This concept of branding things extends even to the level of the commonly-used variable name `pf`, which originated within the original Enzo usage as shorthand for "parameter file," and the name `StaticOutput` as in contrast to

the "streaming" movie format within Enzo. In order to effectively move beyond both astro- and Enzo-centrism, the terminology, attributes, and extensibility of datasets should be emphasized and defined.

17.4.1 Problematic Areas

Attributes on StaticOutput

The following attributes are defined on every `StaticOutput` regardless of whether the dataset is astrophysics, cosmology, or even rectilinear cartesian mesh.

- `current_time` (note: this also is not correctly implemented for Enzo)
- `domain_dimensions`
- `domain_left_edge`
- `domain_right_edge`
- `cosmological_simulation`
- `current_redshift`
- `omega_lambda`
- `omega_matter`
- `hubble_constant`

Even if `cosmological_simulation` is set to off, the cosmology-related parameters will be defined. Additionally, the default "field type" is `gas`, which is globally set and not necessarily trivial to modify. Changing the units to be less astro-specific (which may not be necessary for length units) is part of a larger units-related discussion, rather than part of this YTEP.

Additionally, `StaticOutput` is tied extremely strongly to a file on disk. Because that is largely internally-facing, changing that may not be subject to a YTEP, but rather a simple refactoring.

Finally, not all simulation types have a concept of `domain_dimensions`, even if the indexing system does. This is currently outside the scope of this YTEP. The domain left and right edges also do not always matter for particle simulations (except in non-outflow boundary conditions) but are still always relevant to the indexing system.

Below are a few suggested mechanisms for retaining this information as "first class" attributes of a given data set when appropriate, but to remove it from those datasets where it is not appropriate.

Naming and Branding

Objects will be renamed:

- `StaticOutput` will be renamed to `Dataset`
- `TimeSeriesData` will be renamed to `DatasetSeries` and will no longer exclusively refer to a time-related set of data, but instead include arbitrary collections of datasets.
- Instead of `pf` as shorthand, we will use `ds`.
- Renaming `GeometryHandler` to `Index`

Currently, all datasets expose a `.hierarchy` attribute, shortened to `.h`. This naming is a holdover from the time when Enzo ando ther patch-based AMR datasets were the primary data examined with yt. However, this makes considerably less sense when seen in light of support of particle datasets, semi-structured datasets, unigrid datasets, and eventually unstructured mesh datasets. What we really mean when we say `.hierarchy` or `.h` is *index* or

geometry. Currently, the `StaticOutput` object also possesses a `.geometry` attribute, although this is a string scalar.

I do not think we should replace the `.h` attribute wholesale, and I do not necessarily think that data objects should necessarily directly hang off of the `StaticOutput` (or whatever it is renamed) object. However, I do think that we should eliminate `hierarchy` in favor of something more generic that is more descriptive, and we should consider alternates for creating data objects. Regardless of what we decide on, the `.h` attribute should remain for the time being, and we should also not instantiate our indexing method until requested.

The resolution decided upon during discussion has been:

- Eliminate the `hierarchy` object as a name. `geometry` seems to be the most popular for what the `GeometryHandler` object does.
- Retain the `h` attribute as an alias (for now, possibly forever)
- Each dataset will have an `index` property which will be a `GridIndex`, `OctIndex` etc etc. This is essentially the same as the `Hierarchy` attribute.
- Move data objects up to the top level of `Dataset`.

17.4.2 Domain-Specific Datasets

Because some domains will have fundamental parameters that put into context the data they represent, this YTEP proposes a plugin system wherein domain-specific "contexts" register themselves and specific frontends identify which plugins are applicable to that specific frontend. This dual-ended handshaking helps ensure that plugins ensure they are applicable to a frontend, and that frontends identify potential plugins that work for them.

A domain plugin (called `DomainContext`) will operate *on* a dataset object, adding new attributes, but *not* new methods. This violates common object-oriented philosophy and practice, but from an implementation perspective it seems to be the cleanest and avoiding the most meta-programming.

On instantiation, a static output normally goes through these steps:

1. `_parse_parameter_file`
2. `_setup_coordinate_handler`
3. `_set_units`
4. `_set_derived_attrs`
5. `print_key_parameters`
6. `create_field_info`

This YTEP would propose changing this order to:

1. `_parse_parameter_file`
2. `_setup_coordinate_handler`
3. `_set_units`
4. `_set_derived_attrs`
5. `_apply_domain_contexts`
6. `create_field_info`
7. `print_key_parameters`

_apply_domain_contexts would iterate through the intersecting set of globally and frontend-specific registered domain-specific plugins, and for each one would call the class method: is_appropriate supplying the dataset object (self) as the only argument. If so, the plugin would then return True and an instance of it would be appended to the dataset property domain_contexts (or some other name, as this collides with domain_* referring to simulation spatial information.) Alternately, we could mandate an _adapt_* method (seen below) and in the absence of such a method assume the plugin is blacklisted.

These plugins would then, in sequence, have their apply method called with the dataset as the only argument. They can then add additional attributes to the dataset, as well as additional key parameters to print out. The runtime overhead should be negligible.

This extends further to the compartmentalization of field definitions. We leave that somewhat unspecified here, but domain contexts should enable the application of specific field objects based on runtime parameters. This could mean, for instance, conversion of face-centered to cell-centered quantities, magnetic field analysis, nuclear decay times, and so on. One mechanism for doing this would be to add field objects to the already-created field_info object. (This is why that step must be raised in the list.)

One concern with this is that frontend-specific parameters (i.e., cosmological_simulation) are not universal, so an adapter between the frontend and the plugin needs to be created. We propose that this be required for each frontend by enabling plugins to call methods on the dataset. These methods will be named _adapt_* where the suffix is the contexts's shortname. These will return dictionaries of parameters which will be rigorously checked for contents (i.e., preventing incorrect or incomplete information from being passed back.) Contexts must define these methods.

As an example, here is pseudocode for a cosmological simulation context:

```
class CosmologyContext(DomainContext):
    domain = 'cosmology'

    def __init__(self):
        pass

    @classmethod
    def is_appropriate(cls, pf):
        if not hasattr(pf, '_adapt_cosmology'): return None
        rv = pf._adapt_cosmology()
        if rv['cosmological_simulation'] == 1:
            c = cls()
            return c
        return None

    def apply(self, pf):
        params = pf._adapt_cosmology()
        pf.cosmological_simulation = rv['cosmological_simulation']
        pf.cosmology = Cosmology()
```

This design mechanism is somewhat open for discussion; the problems of adapting varying parameters and matching both the generality of the domain context and the frontend dataset provide challenges. An alternative is to provide a default class method for each context that is used by the base dataset object to obtain a false value.

As noted during discussion, context can and should subclass each other. How this interfaces with which plugin in the order of resolution is not yet clear, as (for instance) the base class should not necessarily modify an attribute when the subclass would then override.

17.4.3 Runtime Extensibility

These domain context will be extensible at runtime by specifying an additional list of plugins to check, by adding additional plugins to the global (and frontend-specific) registry, and by adding to the plugin list for each dataset type.

17.4.4 Implementation

Much of the implementation has been described above. However, these domain plugins should reside in a subdirectory of `data_objects`, specifically named `yt/data_objects/domain_contexts/` and should be limited to one class per file.

17.5 Backwards Compatibility

- The backwards compatibility of renaming is likely quite small, except for those cases where names would be changed.
- The backwards compatibility of checking for `cosmological_simulation` would probably require additional field validation (or instead, fields that are added specifically by the cosmology context).
- Changing `TimeSeriesData` to a new name may need to be gradually introduced, retaining backwards compatibility for a while.
- Fixing Enzo's `current_time` will cause challenges for anyone who is not using internal time conversion factors. I think this number is likely small.

17.6 Alternatives

We could continue with the status quo.

CHAPTER 18

YTEP-0018: Changing dict-like access to Static Output

18.1 Abstract

Created: September 18, 2013 Author: Matthew Turk

Currently, accessing a *StaticOutput* like a dictionary will check the *parameters*, *units*, *time_units* and *conversion_factors* dictionaries. This YTEP proposes changing it such that *no* dictionaries will be queried.

18.2 Status

Proposed

18.3 Project Management Links

There are no easily-identified project management links. However, it should be noted that over the years, numerous times confusion has arisen as to what things like *pf["Time"]* refer to.

18.4 Detailed Description

The conflation of parameters, conversion factors, units and so on causes an enormous amount of confusion. The most common uses of this are:

- Length conversions such as: *1.0/pf['cm']*
- Accessing parameters
- Occasional unit conversions (typically this causes more problems than it solves)

However, the degeneracy that often arises between unit conversions and parameter access is typically quite problematic. This proposes that we simplify the entire procedure to disable all dict-like access, and ensure that individuals access `.parameters` explicitly. This may be unintuitive and will cause large changes to user-facing code, so we may consider re-enabling it.

The difficulty in ensuring that conversions can be conducted in a separate manner arises from the variable conversion factors even within a given frontend; particularly for cosmology simulations, these conversion factors (length, time, etc) change over time.

To implement this, we will ensure that:

1. All places that require a length unit accept a tuple. This is nearly if not completely implemented.
2. For a specified time (until 3.1 is released), dict-like access to the *StaticOutput* object will raise a deprecation warning if the key is not found in *parameters*. This can be elevated to an exception upon request by the user.
3. Ensure the test-suite passes.
4. Update all documentation and examples.

By stepping into this gradually, we will follow the example set forth by the field refactor and enable individuals to see that the behavior is changing without mandating an immediate switch.

18.5 Backwards Compatibility

In 3.0, this will not break scripts; deprecation warnings *will* be issued. In 3.1, this will break a considerable number of scripts that rely on unit conversions mediated by the *StaticOutput* object. This is very worrisome and will require the graduated change to disabling dict-like access.

18.6 Alternatives

We can continue allowing this behavior, but it will continue to cause confusion and impede progress toward a cleaner API.

CHAPTER 19

YTEP-0019: Reduce items in main import

19.1 Abstract

Created: October 2, 2013 Author: Matthew Turk

Currently, `yt.mods` includes a huge number of items, polluting the namespace considerably. Many of these are not necessary, as they are seldom used.

19.2 Status

Proposed

19.3 Project Management Links

19.4 Detailed Description

Currently, the number of items in the `yt.mods` namespace is enourmous. There are 276 items (including module builtins.) By providing a better set of namespaces, we can make all of these items accessible without polluting the namespace itself.

As an example, we should expose the `load` command primarily, and encourage directly importing frontend-specific code if that code needs to be interacted with.

This YTEP concerns two changes in functionality.

- Reduce the number of items in `yt.mods`.
- Make `yt.mods` a superset of functionality of `yt/__init__.py`. This means `startup_tasks.py` (which includes argument parsing, parallel initialization, configuration system reading, and so on.)

This will change some aspects of behavior, as it will make operations that currently require the `ytcfg` variable to be modified *before* any other `startup_tasks` code is executed no longer possible. This number is extremely small, and the primary one is loglevel setting, which is easily exposed in `mylog.setLevel`.

Primarily, we want to make `yt` exist better as a module as well as an environment.

19.4.1 Current Imports

These are the items currently imported, and their proposed status.

We can implement this either through not importing into the main namespace items we do not want to include, or by explicitly enumerating the items to include in an `__all__` attribute.

A few comments on specific items:

- `TransferFunction` classes have been removed and we should investigate other ways of exposing them. One option would be importing the module `transfer_functions` and accessing attributes.
- `HaloFinder` has been removed, so a new method for setting up this process easily and transparently (perhaps with amods) is needed.
- `add_grad` should probably have a new name.

Object	Include
`ARTFieldInfo`	No
`ARTIOFieldInfo`	No
`ARTIOStaticOutput`	No
`ARTStaticOutput`	No
`AnalysisTask`	No
`ArrowCallback`	No
`AthenaFieldInfo`	No
`AthenaStaticOutput`	No
`BinnedProfile1D`	Yes
`BinnedProfile2D`	Yes
`BinnedProfile3D`	Yes
`Camera`	No
`CastroFieldInfo`	No
`CastroStaticOutput`	No
`ChomboFieldInfo`	No
`ChomboStaticOutput`	No
`ClumpContourCallback`	No
`ColorTransferFunction`	No
`ContourCallback`	No
`CoordAxesCallback`	No
`CuttingQuiverCallback`	No
`DummyProgressBar`	No
`EnzoFieldInfo`	No
`EnzoSimulation`	No
`EnzoStaticOutput`	No
`EnzoStaticOutputInMemory`	No
`EnzoTestOutputFileNonExistent`	No
`FLASHFieldInfo`	No
`FLASHStaticOutput`	No
`FieldInfo`	No

Continued on next page

Table 1 – continued from previous page

Object	Include
FixedResolutionBuffer	Yes
FlashRayDataCallback	No
GDFFieldInfo	No
GDFStaticOutput	No
GUIProgressBar	No
GadgetFieldInfo	No
GadgetHDF5StaticOutput	No
GadgetStaticOutput	No
GridBoundaryCallback	No
HaloFinder	No
HomogenizedVolume	No
HopCircleCallback	No
HopParticleCallback	No
ImageArray	Yes
ImageLineCallback	No
InvalidSimulationTimeSeries	No
LabelCallback	No
LinePlotCallback	No
LooseVersion	No
MagFieldCallback	No
MarkerAnnotateCallback	No
MaterialBoundaryCallback	No
MissingParameter	No
MoabFieldInfo	No
MoabHex8StaticOutput	No
MosaicFisheyeCamera	No
NoCUDAException	No
NoStoppingCondition	No
NyxFieldInfo	No
NyxStaticOutput	No
OWLSFieldInfo	No
OWLSStaticOutput	No
ObliqueFixedResolutionBuffer	Yes
OffAxisProjectionPlot	Yes
OffAxisSlicePlot	Yes
OrionFieldInfo	No
OrionStaticOutput	No
ParallelProgressBar	No
ParticleCallback	No
ParticleTrajectoryCollection	Yes
PlanckTransferFunction	No
PlotCallback	No
PlotCollection	Yes
PlotCollectionInteractive	Yes
PlutoFieldInfo	No
PlutoStaticOutput	No
PointAnnotateCallback	No
ProjectionPlot	Yes
ProjectionTransferFunction	No

Continued on next page

Table 1 – continued from previous page

Object	Include
PyneMoabHex8StaticOutput	No
QuiverCallback	No
RAMSESFieldInfo	No
RAMSESStaticOutput	No
SlicePlot	Yes
SphereCallback	No
StreamFieldInfo	No
StreamHandler	No
StreamStaticOutput	No
StreamlineCallback	No
TextLabelCallback	No
TigerFieldInfo	No
TigerStaticOutput	No
TimeSeriesData	Yes
TimestampCallback	No
TipsyFieldInfo	No
TipsyStaticOutput	No
TitleCallback	No
UnitBoundaryCallback	No
ValidateDataField	Yes
ValidateGridType	Yes
ValidateParameter	Yes
ValidateProperty	Yes
ValidateSpatial	Yes
VelocityCallback	Yes
YTAxesNotOrthogonalError	No
YTCannotParseFieldDisplayName	No
YTCannotParseUnitDisplayName	No
YTCloudError	No
YTCoordinateNotImplemented	No
YTCouldNotGenerateField	No
YTDataSelectorNotImplemented	No
YTDomainOverflow	No
YTEllipsoidOrdering	No
YTEmptyClass	No
YTException	No
YTFieldNotFound	No
YTFieldNotParseable	No
YTFieldTypeNotFound	No
YTGeometryNotSupported	No
YTHubRegisterError	No
YTIllDefinedBounds	No
YTIllDefinedFilter	No
YTInvalidWidthError	No
YTNoAPIKey	No
YTNoDataInObjectError	No
YTNoFilenamesMatchPattern	No
YTNoOldAnswer	No
YTNotDeclaredInsideNotebook	No

Continued on next page

Table 1 – continued from previous page

Object	Include
YTNotInsideNotebook	No
YTObjectNotImplemented	No
YTOutputNotIdentified	No
YTParticleDepositionNotImplemented	No
YTRockstarMultiMassNotSupported	No
YTSimulationNotIdentified	No
YTSphereTooSmall	No
YTTooManyVertices	No
YTUnitNotRecognized	No
__builtins__	No
__doc__	No
__file__	No
__level	No
__name__	No
__package__	No
__startup_tasks	No
_fn	No
absolute_import	No
add_art_field	No
add_artio_field	No
add_athena_field	No
add_castro_field	No
add_chombo_field	No
add_enzo_1d_field	No
add_enzo_2d_field	No
add_enzo_field	No
add_field	Yes
add_flash_field	No
add_gadget_field	No
add_gdf_field	No
add_grad	Yes
add_moab_field	No
add_nyx_field	No
add_orion_field	No
add_owls_field	No
add_pluto_field	No
add_quantity	No
add_ramses_field	No
add_stream_field	No
add_tiger_field	No
add_tipsy_field	No
amods	Yes
analysis_task	No
annotate_image	Yes
apply_colormap	Yes
available_analysis_modules	Yes
axis_names	No
bb_apicall	No
cPickle	No

Continued on next page

Table 1 – continued from previous page

Object	Include
callback_registry	No
ceil	No
cls	No
contextlib	No
data_object_registry	No
defaultdict	No
deprecate	No
derived_field	Yes
ensure_dir_exists	No
ensure_list	No
ensure_numpy_array	No
ensure_tuple	No
fix_axis	No
fix_length	No
floor	No
get_available_modules	No
get_hg_version	No
get_image_suffix	No
get_ipython_api_version	No
get_memory_usage	Yes
get_multi_plot	Yes
get_num_threads	No
get_pbar	Yes
get_script_contents	No
get_version_stack	Yes
get_yt_supp	Yes
get_yt_version	Yes
glob	No
humanize_time	No
imgur_upload	No
insert_ipython	Yes
inspect	No
inv_axis_names	No
is_root	Yes
iterable	Yes
just_one	No
load	Yes
load_amr_grids	Yes
load_hexahedral_mesh	Yes
load_particles	Yes
load_uniform_grid	Yes
my_plugin_name	No
mylog	Yes
na	No
name	No
np	Yes
numpy	No
off_axis_projection	Yes
only_on_root	Yes

Continued on next page

Table 1 – continued from previous page

Object	Include
ortho_find	Yes
os	No
parallel_objects	Yes
parallel_profile	Yes
particle_filter	Yes
paste_traceback	No
paste_traceback_detailed	No
pb	No
pdb	No
pdb_run	No
periodic_position	Yes
physical_constants	Yes
print_tb	Yes
projload	No
quantity_info	No
quartiles	Yes
read_struct	No
resource	No
rootloginfo	No
rootonly	Yes
rpdb	No
scale_image	Yes
show_colormaps	Yes
signal	No
signal_ipython	No
signal_print_traceback	No
signal_problem	No
simulation	Yes
struct	No
subprocess	No
sys	No
time	No
time_execution	No
time_function	No
traceback	No
traceback_writer_hook	No
types	No
unparsed_args	Yes
update_hg	No
warnings	No
wraps	No
write_bitmap	Yes
write_fits	Yes
write_image	Yes
write_projection	Yes
x_dict	No
y_dict	No
yt_counters	No
ytcfg	Yes

Continued on next page

Table 1 – continued from previous page

Object	Include
`ytcfgDefaults`	No

19.4.2 Changing `yt/__init__.py` to Import

The second aspect of this YTEP is to change the `yt` module to include everything that is in `yt.mods`, but without the side effects that come from `yt.startup_tasks`. Because importing submodules necessarily will then import `__init__.py`, this means submodules cannot be imported without the whole of `yt` that is exposed in `yt.__init__.py` being imported.

This primarily will affect configuration options, which are largely no longer necessary to modify directly at runtime. Additionally, the old behavior can still be preserved by `yt.mods`.

19.5 Backwards Compatibility

This may break compatibility, although nearly all of the items removed are items that are not typically used in scripts. This list can be modified.

Note that importing frontends into a namespace will still enable them to be used in `load`.

Importing `yt.mods` will still act as before, with option parsing and the like. Importing `yt.config` will result in the config file being parsed once; this means runtime options will need to be modified differently.

19.6 Alternatives

We could identify additional means of reducing the namespace pollution, but this is the main one that I see.

We could also not put anything into `yt/__init__.py`.

CHAPTER 20

YTEP-0020: Removing PlotCollection

20.1 Abstract

Created: March 18, 2014 Author: Matthew Turk

20.2 Status

Completed

20.3 Project Management Links

20.4 Detailed Description

The `PlotCollection` object was designed to work with the package HippoDraw, as a means of controlling multiple plots from a single command. This was also focused very strongly on the idea of viewing a single object from multiple angles, and was mostly useful for my research. All other uses largely used it as the only mechanism of creating plots – not because of any particular functionality it has.

With yt 3.0, I propose that we remove the `PlotCollection` entirely, as its functionality is 100% replaced by the various other objects such as `SlicePlot`, `ProjectionPlot`, `ProfilePlot` and `PhasePlot`, all of which are more modern and provide greater access to the underlying matplotlib objects.

This change will occur in yt 3.0. Nearly all users have migrated to using `SlicePlot` and so on, and we are seeing much greater uptake of `ProfilePlot` and `PhasePlot`. Because we also anticipate growing our community with this release, and because it will be the time when we can break backwards compatibility, this is the most natural time to remove it.

20.5 Backwards Compatibility

Existing scripts that utilize `PlotCollection` will break, but there will be other yt-3.0 changes that they may suffer from anyway.

20.6 Alternatives

I do not think we have the resources to pursue alternate options such as supporting `PlotCollection` in perpetuity.

CHAPTER 21

YTEP-0021: Particle-Only Plots

21.1 Abstract

Created: August 29, 2014 Author: Andrew Myers

This YTEP describes a mechanism for creating scatter plots of particle fields in yt. It was prompted by a question posted to the yt-users list by Jeremy Ritter, linked below. Essentially, it proposes creating a user-facing function called `ParticlePlot` (analagous to `SlicePlot` or `ProfilePlot`) that facilitates plotting arbitrary particle fields against one another.

21.2 Status

Completed

21.3 Project Management Links

- Discussion on yt-users
- Discussion on yt-dev
- Example notebook #1
- Example notebook #2
- Sam's color splatting PR

21.4 Detailed Description

Currently, to make plots like those in the linked notebooks, you would have to grab the particle data from the data source and feed them to something like pyplot.plot(). Instead of units, labels, and log_scales getting grabbed from

the FieldInfoContainer, they would need to be set up manually. Furthermore, the standardized interface for modifying yt plots that exists in PlotWindow would not be available.

Instead, we could create a ParticlePlot class that would act like the currently existing yt plotting classes. The constructor would take:

- data_source: an AMR3DData object
- x_fields: str or list, the field(s) to put on the x-axis
- y_fields: str or list, same but for the y-axis
- color: either a color string, or another particle field to be mapped to a color scale

If x_fields and y_fields are strings, this would add a single scatter plot to the ParticlePlot. If they are lists of field_names, then a series of plots will be added, in the style of (for example) PhasePlot. The standard methods for modifying these plots (e.g. set_log, set_units, set_cmap, etc.) should all work as expected, and they should be able to be saved / sent to the notebook as normal.

21.5 Implementation Details

My current implementation wraps pyplot.plot(), but because pyplot.plot can be slow when the number of points is large, ParticlePlot should instead use Sam Skillman's particle splatting code to create something like a FixedResolutionBuffer. This could then be displayed with pyplot.imshow(). This would also make it easy for users to access the raw image and pass it to another plotting routine, if they prefer.

21.6 Inheritance Structure

ParticlePlot shares a lot of its functionality with PhasePlot, so the implementation should be similar. In particular, ParticlePlot should inherit from ImagePlotContainer and the individual plots in it should be ParticlePlotMPL objects that inherit from ImagePlotMPL.

21.7 Open issues

Plots that have spatial variables on both axes are logically different from those that don't in a few ways. For instance, it could be misleading if the aspect ratios are different for two spatial axes, as in the second linked notebook. Also, things like "pan" and "zoom" make sense for spatial data, but not when plotting, say, velocity versus position. Should we handle spatial plots differently, as Nathan suggested in the yt-dev discussion above? Spatial plots could in inherit from PlotWindow to take advantage of all the methods and callbacks in there.

21.8 Backwards Compatibility

None; all existing code should work exactly as before.

21.9 Alternatives

Alternatively, there could be no mechanism for making particle scatter plots inside of yt, and users could call pyplot.plot() or whatever directly.

CHAPTER 22

YTEP-0022: Benchmarks

22.1 Abstract

Created: January 19, 2015 Author: Matthew Turk

This document proposes a mechanism for tracking performance improvements and regressions, based on the airspeed velocity project for automated benchmarking.

22.2 Status

Proposed

22.3 Project Management Links

- Pull request: https://bitbucket.org/yt_analysis/yt/pull-request/1415/add-airspeed-velocity-config-file/
- ASV homepage: http://spacetelescope.github.io/asv/
- Intro talk: http://youtube.com/watch?v=OsxJ5O6h8s0

22.4 Detailed Description

22.4.1 Background

Particularly during the transition from yt 2 to yt 3, there has been a large degradation in overall performance. The underlying implementation of data selection has become considerably more general and the array operations have all been overloaded, both of which have performance impacts, both of which are large improvements in functionality and usability. However, in order to address the sad and unacceptable performance degradations, this YTEP has been proposed as the first step toward tracking performance and attempting to mitigate regressions now and in the future.

22.4.2 Proposed Actions

Currently, as described in *YTEP-0007: Automatic Pull Requests' validation*, we conduct automatic validation of pull requests. A mechanism for examining and validating change in performance should also be implemented.

The "airspeed velocity" project has been designed to track the changes in performance as denoted by specific benchmarks over time in a given project. What we will do is implement a number of characteristic benchmarks, potentially relying on larger datasets than we do for simple testing, and track how changes to the code make these benchmarks run faster or slower. As part of the CI process, these results will be posted to a website (ASV has the ability to generate such websites, along with detailed drilldowns into specific routines, automatically.) Because of how asv works, these updates will likely need to be a part of the post-acceptance process, rather than pre-acceptance.

A sample website can be seen here: http://mdboom.github.io/astropy-benchmark/

There are three axes of benchmarks, along with the specific benchmark being run.

- Changeset hash being benchmarked
- Version of external dependencies
- Machine on which benchmarks are run

This YTEP proposes:

- Adding running the benchmarks to the CI suite
- Adding publishing of the benchmarking results to the CI suite (perhaps at speed.yt-project.org)
- Encouraging writing new benchmarks by project contributors

Once a benchmark has been written, it can be used forevermore, and even evaluated against past changeset hashes in the yt repository. Writing benchmarks is easy, too – easier even than tests.

22.4.3 Proposed Repository Layout

Keeping benchmarks next to the code is important to avoid fragmentation. However, the repository containing the benchmark results will balloon in size, as results and outputs will be committed with mildly reckless abandon. So this YTEP proposes two repositories:

- Existing yt repository, which will contain the asv configuration file and the benchmarks directory
- A `yt-benchmarks` repository, which will contain the results (committed, included in the repo, and probably gigantic) along with a bootstrap script that makes symlinks to the asv configuration file and the benchmarks directory.

This way, we can very easily run in this directory, commit, and auto-publish. It will mean that all the benchmarks are in the main repository, for ease of contribution and modification, but all the results are stored elsewhere. It also will make it easier for folks who want to run and store results on other machines to do so.

22.5 Backwards Compatibility

No backwards compatibility problems exist.

22.6 Alternatives

We could implement our own framework, but this one exists and is pretty nice.

CHAPTER 23

YTEP-0023: yt Community Code of Conduct

23.1 Abstract

Created: July 11, 2015

Author: Britton Smith

This document contains the code of conduct for the yt community. It is a near exact copy of the Astropy Community Code of Conduct, except that we will employ our own confidential email address for community members to report violations.

23.2 Status

Completed

23.3 Project Management Links

- Astropy Communitiy Code of Conduct

23.4 Detailed Description

The code of conduct, whose language is below, will be displayed in the following places:

- Developer Documentation
- yt Project About page
- yt Project Community page
- yt Project Development page

Emails sent to the confidential address will be seen by Hilary Egan, Britton Smith, and John Zuhone.

23.5 yt Community Code of Conduct

The community of participants in open source Scientific projects is made up of members from around the globe with a diverse set of skills, personalities, and experiences. It is through these differences that our community experiences success and continued growth. We expect everyone in our community to follow these guidelines when interacting with others both inside and outside of our community. Our goal is to keep ours a positive, inclusive, successful, and growing community.

As members of the community,

- We pledge to treat all people with respect and provide a harassment- and bullying-free environment, regardless of sex, sexual orientation and/or gender identity, disability, physical appearance, body size, race, nationality, ethnicity, and religion. In particular, sexual language and imagery, sexist, racist, or otherwise exclusionary jokes are not appropriate.
- We pledge to respect the work of others by recognizing acknowledgment/citation requests of original authors. As authors, we pledge to be explicit about how we want our own work to be cited or acknowledged.
- We pledge to welcome those interested in joining the community, and realize that including people with a variety of opinions and backgrounds will only serve to enrich our community. In particular, discussions relating to pros/cons of various technologies, programming languages, and so on are welcome, but these should be done with respect, taking proactive measure to ensure that all participants are heard and feel confident that they can freely express their opinions.
- We pledge to welcome questions and answer them respectfully, paying particular attention to those new to the community. We pledge to provide respectful criticisms and feedback in forums, especially in discussion threads resulting from code contributions.
- We pledge to be conscientious of the perceptions of the wider community and to respond to criticism respectfully. We will strive to model behaviors that encourage productive debate and disagreement, both within our community and where we are criticized. We will treat those outside our community with the same respect as people within our community.
- We pledge to help the entire community follow the code of conduct, and to not remain silent when we see violations of the code of conduct. We will take action when members of our community violate this code such as contacting confidential@yt-project.org (all emails sent to this address will be treated with the strictest confidence) or talking privately with the person.

This code of conduct applies to all community situations online and offline, including mailing lists, forums, social media, conferences, meetings, associated social events, and one-to-one interactions.

The yt Community Code of Conduct was adapted from the Astropy Community Code of Conduct, which was partially inspired by the PSF code of conduct.

23.6 Alternatives

None.

CHAPTER 24

YTEP-0024: Alternative Smoothing Kernels

24.1 Abstract

Created: August 1, 2015 Author: Bili Dong

This YTEP proposes to add alternative smoothing kernels besides the current standard cubic spline one, make them available to the smoothing operations, and define a convenient interface for users to choose among them.

24.2 Status

Completed

24.3 Project Management Links

- yt-dev thread
- yt PR #1670
- yt PR #1712
- yt PR #1830
- ytep PR #53

24.4 Detailed Description

Currently in yt, the standard cubic spline kernel is exclusively used for smoothing operations. It is a desired feature for yt to have support for varied smoothing kernels.

The implementations of the kernel functions themselves are straightforward. [DA2012] is referenced for the function forms and the kernel names.

The bigger challenge is the design of a convenient user interface for future users and a convenient application programming interface for future developers. Details of those designs are explained in the following sections.

24.4.1 User Interface

Future users can specify which kernel to use in the smoothing operations by passing a keyword argument `kernel_name` to the relevant functions. Currently, functions with potential access to kernels include `Dataset.add_deposited_particle_field` and `Dataset.add_smoothed_particle_field`[1]. The naming scheme for fields added through these functions is an extension of the scheme described in SPH Fields, which is proposed by Nathan Goldbaum in this yt-dev thread. So a particle field `("particletype", "fieldname")` smoothed by a certain kernel can be accessed by `("deposit", "particletype_kernelname_smoothed_fieldname")`, except for the cubic spline kernel, whose smoothed field remains to be `("deposit", "particletype_smoothed_fieldname")` (the same as before).

For example, as demonstrated by the following code, the particle field `("PartType0", "Density")` is smoothed using a quintic kernel and the resultant smoothed field could be accessed by `("deposit", "PartType0_quintic_smoothed_Density")`.[2]

```
import yt

ds = yt.load("GadgetDiskGalaxy/snapshot_200.hdf5")
ds.add_smoothed_particle_field(("PartType0", "Density"), kernel_name="quintic")

yt.ProjectionPlot(ds, "z", ("deposit", "PartType0_quintic_smoothed_Density"))
```

Below is a table of available `kernel_name` and the corresponding name in [DA2012]. All kernels are in 3D (a.k.a. $\nu = 3$).

Table 1: Kernel Names

kernel_name	Name in [DA2012]
cubic	Cubic spline
quartic	Quartic spline
quintic	Quintic spline
wendland2	Wendland C^2
wendland4	Wendland C^4
wendland6	Wendland C^6

24.4.2 Application Programming Interface

When a kernel function is needed, `get_kernel_func` is used to retrieve it. Given the string of the kernel name, a kernel function of the type `kernel_func` is returned. Both `get_kernel_func` and `kernel_func` are defined in `geometry/particle_deposit.pxd`. The following snippet demonstrate their usage, assuming the source file is in the same directory as `particle_deposit.pxd`.

[1] `Dataset.add_smoothed_particle_field` is a wrapper of `add_volume_weighted_smoothed_field`. It is more convenient to use. So, for simplicity, `add_volume_weighted_smoothed_field` will be omitted in the following discussions.

[2] The dataset can be downloaded from here.

```
from .particle_deposit cimport kernel_func, get_kernel_func

cdef class DemoParticleSmoothOperation:
    cdef kernel_func sph_kernel
    def __init__(self, kernel_name):
        self.sph_kernel = get_kernel_func(kernel_name)
```

Once the kernel function is retrieved, `self.sph_kernel` could be utilized to do the smoothing.

The rest of the changes to the API is merely the passing of the keyword argument `kernel_name`. Below is a table demonstrating the potential passing routes:

Table 2: Passing of `kernel_name` through Methods (or Functions)

#	Method	Pass to[3]	File Path
1	`SimpleSmooth` (aliased as `deposit_simple_smooth`)[4]		`yt/geometry/particle_deposit.pyx`
2	`VolumeWeightedSmooth` (aliased as `volume_weighted_smooth`)[4]		`yt/geometry/particle_smooth.pyx`
3	`SmoothedDensityEstimate` (aliased as `density_smooth`)[4]		`yt/geometry/particle_smooth.pyx`
4	`ARTIORootMeshSubset.deposit`	1	`yt/frontends/artio/data_structures.py`
5	`YTCoveringGridBase.deposit`	1	`yt/data_objects/construction_data_containers.py`
6	`AMRGridPatch.deposit`	1	`yt/data_objects/grid_patch.py`
7	`UnstructuredMesh.deposit`	1	`yt/data_objects/unstructured_mesh.py`
8	`OctreeSubset.deposit`	1	`yt/data_objects/octree_subset.py`
9	`OctreeSubset.smooth`	2, 3	`yt/data_objects/octree_subset.py`
10	`OctreeSubset.particle_operation`	2, 3	`yt/data_objects/octree_subset.py`
11	`Dataset.add_deposited_particle_field`	4 - 8	`yt/data_objects/static_output.py`
12	`Dataset.add_smoothed_particle_field`	9	`yt/data_objects/static_output.py`

To demonstrate how 4 - 10 utilize 1 - 3, the main structure of the `smooth` method is shown below (irrelevant parts are ignored; `deposit` and `particle_operation` are similar).

```
def smooth(self, method = None, kernel_name = "cubic", ...):
    cls = getattr(particle_smooth, "%s_smooth" % method, None)
    op = cls(..., kernel_name)
```

`op` is used for the actual smoothing operations thereafter.

For 11 & 12, they simply call the dataset's `deposit` or `smooth` method to get the smoothing operations done.

[3] This column indicates the possibility that `kernel_name` could be passed to 'Pass to', which also depends on another parameter `method`.
[4] When a class is given, its `__init__` method is meant.

24.4.3 Reference

24.5 Backwards Compatibility

New functionality is accessed by the keyword argument `kernel_name` with default value `kernel_name = "cubic"`, so existing codes' behavior won't change.

24.6 Alternatives

None.

CHAPTER 25

YTEP-0025: The ytdata Frontend

25.1 Abstract

Created: August 31, 2015 Author: Britton Smith

This YTEP proposes to make data products created by yt into loadable datasets. Primarily, this will provide the following features:

- exporting geometric data containers to datasets that can be reloaded for further geometric selection and analysis.
- exporting plot-type data (projections, slices, profiles) so that they can be moved, reloaded, and manipulated to make new images.

25.2 Status

Completed

25.3 Project Management Links

- yt PR #1718: the accepted pull request containing the full implementation

25.4 Detailed Description

Currently, yt's main data products (data containers, projections, slices, profiles) can only be used with their full functionality with the original dataset loaded. This is cumbersome when the datasets are so large that they can only be hosted at remote facilities. Creating publication-quality images from such data either requires a cycle of tweaking, transferring, viewing, and cursing or creating custom intermediate data products and plotting codes.

This YTEP proposes to create functionality that will allow for the above data products to be exported to a format that can be reloaded as a full-fledged dataset.

The proposed functionality consists of two main components: functionality to save objects to disk and a frontend responsible for reloading the saved objects.

25.4.1 Exporting

A general function for saving array data associated with an open dataset will be responsible for writing data to disk. Data will be written to a single hdf5 file. Metadata associated with the dataset (i.e., current_time, current_redshift, cosmological parameters, domain dimensions) will be saved as attributes of the root file group. By default, data will be saved to a "grid" group with "units" attributes saved for each dataset. This function is implemented as `save_as_dataset` in `yt/frontends/ytdata/utilities.py` and imported in the main yt import.

The above function will be called by the user-facing functions, `YTDataContainer.save_as_dataset`, `ProfileND.save_as_dataset`, and `FixedResolutionBuffer.save_as_dataset`, which will optionally take a filename and a list of fields. If no field list is given, then the fields that have been queried and cached will be saved. This function will also make sure that fields necessary for geometric selection (grid position/cell size, particle position) are also saved. Mesh data will be saved to the "grid" group and particle data will be saved to groups named after the specific particle type.

25.4.2 ytdata Frontend

This frontend will be responsible for reloading all data saved with the above method. As this data is of multiple types, this will actually be multiple frontends living under the general "ytdata" heading. All dataset types will inherit from the `YTDataset` class. See *ytdata Dataset Types* for a description of each class. For each loaded dataset, `ds`, in the ytdata frontend, `ds.data` will provide direct access to the field data.

Geometrical Data Containers

Fields queried from data containers are returned as unordered, one-dimension arrays and, thus, most closely resemble particle datasets. All geometric data containers are reloaded as type `YTDataContainerDataset`, which is a particle dataset type. Mesh data is stored with the corresponding `dx`, `dy`, and `dz` fields such that derived fields like `cell_volume` can be created. All mesh data is aliased to the "gas" field type. The `data` attribute associated with the loaded dataset will be a data container configured identically to the original data container. In the case of ray data containers, this is not possible as a ray is defined by cells it intersects and not cells/particles enclosed within. In this case, `data` will be an instance of `ds.all_data()`. Field access through conventional data containers is also possible.

```
ds = yt.load("enzo_tiny_cosmology/DD0046/DD0046")

sphere = ds.sphere([0.5]*3, (10, "Mpc"))
sphere.save_as_dataset(fields=["density", "particle_mass"])

sds = yt.load("DD0046_sphere.h5")

# sphere with the same center and radius
print (sds.data)
print (sds.data["grid", "density"])
print (sds.data["gas", "density"])
print (sds.data["all", "particle_mass"])
print (sds.data["all", "particle_position_x"])
```

(continues on next page)

(continued from previous page)

```
# create a data container
ad = sds.all_data()
print (ad["grid", "density"])
print (ad["all", "particle_mass"])
```

Grid Data Containers

Covering grids, smoothed covering grids, and arbitrary grids return 3D arrays and so can be treated as uniform grid datasets. After being saved with `save_as_dataset`, these are reloaded as type `YTGridDataset`, which is a uniform grid that also supports particles. `FixedResolutionBuffer` objects saved with `save_as_dataset` will be reloaded as this type as well, only 2D. In this case, `ds.data` will give access to the multi-dimensional field arrays.

```
ds = yt.load("enzo_tiny_cosmology/DD0046/DD0046")

cg = ds.covering_grid(level=0, left_edge=[0.25]*3, dims=[16]*3)
cg.save_as_dataset("cg.h5", ["density", "particle_mass"])
cg_ds = yt.load("cg.h5")

# this has the dimensions of the original covering grid
print (cg_ds.data["gas", "density"]).shape

# access via geometric selection
ad = cg_ds.all_data()
print (ad["gas", "density"])
print (ad["all", "particle_mass"])

ray = cg_ds.ray(cg_ds.domain_left_edge, cg_ds.domain_right_edge)
print (ray["gas", "density"])

# FRBs
proj = ds.proj("density", "x", weight_field="density")
frb = proj.to_frb(1.0, (800, 800))
frb.save_as_dataset(fields=["density"])
fds = yt.load("DD0046_proj_frb.h5")
print (fds.data["density"])
```

Projections and Slices

Projections and slices are like two-dimensional particle datasets where the x and y fields are "px" and "py". They are reloaded as type `YTProjectionDataset`, which is a subclass of `YTDataContainerDataset`. Reloaded projection or slice data can be selected geometrically or fed into a `ProjectionPlot` or `SlicePlot`. In these cases, `ds.data` is an instance of `ds.all_data()`.

```
ds = yt.load("enzo_tiny_cosmology/DD0046/DD0046")

proj = ds.proj("density", "x", weight_field="density")
proj.save_as_dataset("proj.h5")

gds = yt.load("proj.h5")
print (gds.data["gas", "density"])
p = yt.ProjectionPlot(gds, "x", "density", weight_field="density")
p.save()
```

The above would enable someone to make projections or slices of large datasets remotely, then download the exported dataset, and perfect the final image on a local machine. On and off-axis slices are implemented. Off-axis projections are not implemented at this time as they use totally different machinery. In this case, the best strategy would be to create an FRB and call `save_as_dataset` on that.

General Array Data

Array data written with the base `save_as_dataset` function can be reloaded as a non-spatial dataset. Geometric selection is not possible, but the data can be accessed through the `YTNonspatialGrid` object, `ds.data`. This object will only grab data from the hdf5 file and do further selection on it.

```
from yt.frontends.ytdata.api import save_as_dataset

ds = yt.load("enzo_tiny_cosmology/DD0046/DD0046")

region = ds.box([0.25]*3, [0.75]*3)
sphere = ds.sphere(ds.domain_center, (10, "Mpc"))

my_data = {}
my_data["region_density"] = region["density"]
my_data["sphere_density"] = sphere["density"]
save_as_dataset(ds, "test_data.h5", my_data)

ads = yt.load("test_data.h5")
print (ads.data["region_density"])
print (ads.data["sphere_density"])
```

Profiles

1, 2, and 3D profiles are like 1, 2, and 3D uniform grid datasets where dx, dy, and dz are different and have different dimensions. `YTProfileDataset` objects inherit from the `YTNonspatialDataset` class. Similarly, the data can be accessed from `ds.data`. The x and y bins will be saved as 1D fields and fields named after the x and y bin field names will be saved with the same shape as the actual profile data. This will allow for easy array slicing of the profile based on the bin fields.

```
ds = yt.load("enzo_tiny_cosmology/DD0046/DD0046")
profile = yt.create_profile(ds.all_data(), ["density", "temperature"],
                            "cell_mass", weight_field=None)
profile.save_as_dataset()

pds = yt.load("DD0046_profile.h5")
# print the profile data
print pds.data["cell_mass"]
# print the x and y bins
print pds.data["x"], pds.data["y"]
# bin data shaped like the profile
print pds.data["density"]
print pds.data["temperature"]
```

25.4.3 ytdata Dataset Types

Name	Inheritance	Purpose	Dataset Type	Geometric Selection
`YTDataset`	`Dataset`	common functionality for other dataset types	n/a	n/a
`YTDataContainerDataset`	`YTDataset`	geometric data containers (sphere, region, ray, disk)	particle	yes
`YTSpatialPlotDataset`	`YTDataContainerDataset`	projections, slices, cutting planes	particle	yes
`YTGridDataset`	`YTDataset`	covering grids, arbitrary grids, fixed resolution buffers	grid w/particles	yes
`YTNonspatialDataset`	`YTGridDataset`	general array data	grid	no
`YTProfileDataset`	`YTNonspatialDataset`	1-, 2, and 3D profiles	grid	no

25.5 Backwards Compatibility

Currently, the only API breakage is in the `AbsorptionSpectrum`. Previously, it accepted a generic hdf5 file created by the `LightRay`. As per the open PR, the `LightRay` now writes out a yt.loadable dataset that is loaded by the `AbsorptionSpectrum`.

Other than the above, this is all new functionality and so has no backward incompatibility. One general change made to the yt codebase is that places that refer to `index` fields (`x`, `y`, `z`, `dx`, etc.) now refer to `(<fluid_type>, "dx")` instead of `("index", "dx")`. This is to allow fields like `cell_volume` to be created from the `("grid", "dx")` field that, for the ytdata frontend, lives on disk instead of the version being generated by the geometry handler. For actual grid datasets, we simply create an alias from `(<fluid_type>, "dx")` to `("index", "dx")` upon loading. This should be completely transparent to the user.

25.6 Alternatives

We could create custom binary files for every type of plot and data container. We could also revive the concept of saving pickled objects that was used somewhat in yt-2.

CHAPTER 26

YTEP-0026: NumPy-like Operations

26.1 Abstract

Created: September 21, 2015

Author: Matthew Turk

This YTEP describes implementing some NumPy-like and potentially some Pandas-like operations on data container objects.

26.2 Status

This YTEP is proposed, but proof-of-concept code has been developed and issued in a PR: https://bitbucket.org/yt_analysis/yt/pull-requests/1763

Once the YTEP PR has been accepted, documentation will be added to the PR to the codebase.

26.3 Project Management Links

Any external links to:

- PR with first work-in-progress: https://bitbucket.org/yt_analysis/yt/pull-requests/1763

26.4 Detailed Description

26.4.1 Background

Data objects in yt are lazy-loaded; only when data is accessed is it read from disk. However, the way they behave is similar to "data frames" or numpy named dtypes – they act as though they are dicts-of-arrays, with some operations

being defined that operate in parallel-aware ways.

However, this is something of a leaky abstraction; in order to compute relatively simple operations, the *.quantities* object has to be accessed, the correct "quantity" to use determined, and then called.

But, many of these quantities map relatively simply to NumPy operations.

This YTEP doesn't (yet) address adding other, Pandas-like operations (such as select or group) even though they also map to yt operations; that may come in the future.

26.4.2 What Can Be Done

I think we should map numpy array operations to quantities and other things! And while we're at it, let's add on very simple "plot" operations. Furthermore, to make the connection more explicit, slices will be implemented as well to generate data objects and selections. The dataset object will have a *.r* attribute, aliased to the much more descriptive `.region_expression`, which enables directly slicing it, which will either return a region, a slice, or an `arbitrary_grid`, depending on how many dimensions are used and if an imaginary step is supplied (like `np.mgrid`). This will accept a unitful slice.

26.4.3 Implementation

This will be implemented very simply as a set of aliases that look at the input arguments and then generate results from them.

NumPy arrays have several operations that return scalars, which is what we want to map to within these operations:

- `all`
- `any`
- `argmax`
- `argmin`
- `max`
- `mean`
- `min`
- `prod`
- `ptp`
- `std`
- `sum`
- `var`

For the purposes of this YTEP, we will concern ourselves with `argmax`, `argmin`, `max`, `mean`, `min`, `std`, `ptp`, `sum`, and also the non-NumPy operations `hist` and `integrate`, which normally do not return a single scalar but a set that does not correspond to the number of elements in the array.

We break these up based on the axis argument, and other optional arguments. Below is the enumerated behavior. Note that for those items that can be computed in a single pass (i.e., statistical information about the fields as a whole) we will likely implement a system that computes them in a single pass and caches them, so that `min` and `max` and `std` will cache in-between calls and only require a single pass over the array.

argmax

The mandatory argument is the field over which the maximum is to be computed; the default return argument is the index, but the `axis` optional parameter can specify one or more fields that will be returned. (For instance, one could supply `('x', 'y', 'z')` and be handed back the spatial locations.

argmin

The mandatory argument is the field over which the minimum is to be computed; the default return argument is the index, but the `axis` optional parameter can specify one or more fields that will be returned. (For instance, one could supply `('x', 'y', 'z')` and be handed back the spatial locations.

max

The mandatory argument is the field of which the maximum is to be computed. This can be a list of fields.

This accepts the optional argument `axis`. If `axis` is a spatial axis (as defined by `coordinates.axis_names` and thus including 0, 1, 2, and the axis names) it will generate a *maximum intensity projection* along that axis of the specified field.

mean

The mandatory argument is the field to average. This will return either a projection if the axis is spatial, or a quantity result.

The optional `axis` argument can either be the spatial axis along which the weighted projection can be computed (defaults to weighted by `ones`, which is usually not desired for astro data, but may be for other data) or `None`. Non-spatial axes are not supported.

The optional `weight` argument (which defaults to `ones`) describes how to weight this average. If `axis` is `None` and `weight` is `None`, it will compute the sum; if `axis` is None and `weight` is not `None`, it will compute the `weighted_average_quantity`.

min

The mandatory argument is the field of which the minimum is to be computed. This can be a list of fields.

Because we do not have "minimum intensity projections," spatial axes are not supported.

std

The mandatory argument is the field of which the standard deviation is to be computed. This can be a list of fields.

The optional argument `weight` will describe the weight for computing standard deviation.

ptp

The mandatory argument is the field of which the peak-to-peak is computed.

sum

The mandatory argument is the field to sum.

The `axis` argument, if spatial, will be the axis along which the projection will be taken. This must either be `None` or a spatial axis. The weighting will be `None`, and thus it will be the line integral. (Note that this will *not* includes a `dl` term, as it will be using the `sum` method.)

integrate

The mandatory `field` argument is the field to integrate; if `axis` is one of the coordinate axes, the return value will be a projection. This will be using the standard projection method, which includes `dl`.

If the `axis` argument is not a spatial dimension, maybe it could return a profile of some type? I'm not sure.

hist

This should return a profile. Determining the most natural way to map how we profile (i.e., the fields along the axes, and the weighting) is an open question. But, it seems to me that we want to do something like:

- Mandatory argument: field or fields to take the average of, or the sum of. If bins is not specified, the returned profile will compute the sum of this field in bins along the x axis; this is somewhat of a weird conditional, but seems to match the closest.
- Optional `weight` argument: the field to use as the weight; if not specified, this will just be a sum.
- Optional `bins` argument: the x and optionally y field to use as bins

__getitem__

The slice operation on a shadow `.r` quantity should return regions or slices.

If one axis is fully-specific, it will be the slice along that axis. If all three are left as start/stop tuples *with no step*, it will be a region. These can be either float values or unitful objects or tuples of `(val, unit_name)`.

If a step is supplied, it will need to be supplied for all three dimensions, will need to be imaginary (i.e., `64j`) and it will be interpreted as input to an `arbitrary_grid` object. The start/stop will provide the left and right edges and the step will provide the number of dimensions.

plot

The `plot` operation will only be implemented on things that have obvious plotting candidates – slices, projections, profiles. This will default to creating the necessary `PlotWindow` or related class, and will try to choose sane defaults for it. For instance, this could wrap `to_pw`. In contrast to `to_pw`, this will also default to *native* plot coordinates, as we want this to match more closely the behavior that would be done by simply plotting the field.

26.4.4 Examples

At the present to get a projection plot of a data object, one would do:

```
obj = ds.sphere((100, 'cm'), 'c')
p = yt.ProjectionPlot(ds, 'x', 'density', data_source = obj)
p.show()
```

or:

```
obj = ds.sphere((100, 'cm'), 'c')
proj = ds.proj("x", "density", data_source=obj)
p = proj.to_pw()
p.show()
```

The alternate here would be:

```
obj = ds.sphere((100, 'cm'), 'c')
p = obj.sum("density", axis="x")
p.plot()
```

The histogram could be computed:

```
obj = ds.sphere((100, 'cm'), 'c')
p = obj.hist("density", bins="temperature", weight="cell_mass")
p.plot()
```

The slicing would look like:

```
ds = yt.load("galaxy0030")
my_obj = ds.r[(100,'kpc'):(200,'kpc'), :, (100,'kpc'):(200,'kpc')]
```

The way to construct this at present would be, which is a bit cumbersome (there are other ways to do this, too, but this is the one that is the clearest):

```
ds = yt.load("galaxy0030")
left_edge = ds.domain_left_edge.in_units("kpc").copy()
left_edge[0] = 100
left_edge[2] = 100
right_edge = ds.domain_right_edge.in_units("kpc").copy()
right_edge[0] = 200
right_edge[2] = 200
center = (left_edge + right_edge)/2.0
my_obj = ds.region(center, left_edge, right_edge)
```

Or for a slice:

```
ds = yt.load("galaxy0030")
my_obj = ds.r[(100,'kpc'):(200,'kpc'), (250,'kpc'), (100,'kpc'):(200,'kpc')]
my_obj.plot()
```

At present, we would have to:

```
ds = yt.load("galaxy0030")
left_edge = ds.domain_left_edge.in_units("kpc").copy()
left_edge[0] = 100
left_edge[2] = 100
right_edge = ds.domain_right_edge.in_units("kpc").copy()
right_edge[0] = 200
right_edge[2] = 200
center = (left_edge + right_edge)/2.0
reg = ds.region(center, left_edge, right_edge)
my_obj = ds.slice(1, (250,'kpc'))
my_obj.to_pw("density")
```

Another example is how to make very terse computations, which still demonstrate reasonably clearly what they do:

```
ds = yt.load("IsolatedGalaxy/galaxy0030/galaxy0030")
dd = ds.r[:,:,:]
print dd.mean(["velocity_%s" % ax for ax in 'xyz'], weight="cell_mass")
```

This returns::

```
[37021.0582639 cm/s, 35794.630883 cm/s, 82204.2708063 cm/s]
```

Note that we can also do:

```
print ds.r[:,:,:].mean(["velocity_%s" % ax for ax in 'xyz'], weight="cell_mass")
```

With the step functionality, this is also possible:

```
g = ds.r[::128j,::128j,::128j]
g["density"]
```

which will be an `arbitrary_grid` object with 128 cells in each dimension.

We may at some point want to add pandas-like selection and indexing functions (http://pandas.pydata.org/pandas-docs/stable/indexing.html) but right now the use case is less clear. Maybe having select() be an alias for cut_region, or adding in a groupby method (maybe; not sure that's useful unless it were by binning) would be interesting, but not immediately clear to me.

This work, if completed, will include an overhaul of the documentation to reflect this, as I think it is considerably terser and more expressive.

26.5 Backwards Compatibility

There are no backwards-compatible issues.

26.6 Alternatives

I do not know if there are alternatives to consider; in many ways, this will open us up to more straightforward utilization of tools like `xray` and `dask`.

CHAPTER 27

YTEP-0027: Non-Spatial Data

27.1 Abstract

Created: December 1, 2015 Author: Matthew Turk, Nathan Goldbaum, John ZuHone

This YTEP outlines a plan to implement support for native non-spatial data representations in yt.

27.2 Status

In Progress

27.3 Project Management Links

- This pull request is the first attempt at implementing this: https://bitbucket.org/yt_analysis/yt/pull-requests/1891/wip-supporting-non-spatial-coordinate
- This Trello card discusses it a bit: https://trello.com/c/7d5PCUym/7-index-arrays as does this one: https://trello.com/c/MXF1sWam/6-non-spatial-data

27.4 Detailed Description

27.4.1 Background

Currently, most of yt assumes that its data structures (particularly for purposes of selection and units) are related to spatial coordinates. This leads to issues such as spherical and cylindrical coordinates believing their angular coordinates are in `code_length`, having to pretend that pressure coordinates are `code_length`, and so on.

An additional complication is that at present, index operations (particularly in selection operations) cannot know in advance that their input arrays are in "index space." This leads to costly operations that check the units (which are

assumed to be `code_length`) and converts if need be. It is often very difficult to create a situation where the arrays are not in those units, though.

Fortunately, there are very few places where the arrays used to index the dataset are utilized directly; for the most part, they are manually stripped of units and then re-applied with the correct units in classes such as the spherical coordinates handler.

This YTEP concerns itself with a few things:

- Allowing datasets to be loaded that are indexed in non-spatial dimensions (for instance, lat, lon, pressure)
- Developing unitful coordinate systems for these non-spatial datasets
- Implementing a custom coordinate handler

27.4.2 Why is this hard?

There are assumptions made in a number of places that data is spatial. Often this shows up in one of these ways:

- Calls to `ensure_code` or conversions explicitly to `code_length`.
- Assumptions that a set of units can be represented as a form of length, for instance during integration.
- Inhomogeneous units in a single YTArray are not supported in the current development tip of yt. Some behavior can be mocked up using object arrays, but this is incredibly unreliable.

27.4.3 Implementation of Index Arrays

To address this issue, an implementation of an object explicitly for indexing data has been created, currently called an `IndexArray`. This object subclasses from `YTArray`, but differs in some crucial ways.

- Multiple units may be specified. These units must be of the same length as the final axis of the array.
- The units in an array are immutable. To change units, the array must be copied. Practically, this means that `convert_to_units` will raise an exception, but it brings with it the benefit that it is difficult to find oneself in a situation where something like `domain_left_edge` is not the native units of the indexing system.
- Fancy-indexing is not possible; only slicing can be conducted.

These arrays are almost always assumed to be created internally within yt. Some situations, such as specifying a "center" to an object, can accept `IndexArray` objects.

27.4.4 Implementation of Coordinates

For inhomogeneous units to be useful, there must be a mechanism for specifying the units to a coordinate handler. The implementation of a `CustomCoordinateHandler` manages this task. This coordinate handler assumes that the coordinate space is functionally Cartesian, but where the axes correspond to non-spatial information. For instance, you might have the first axis be mass, the second time, the third distance.

> **Warning:** At present, distance metrics are *assumed* to be scaled identically amongst the three axes. This means that distance is computed in a Euclidean fashion!

To specify this, the `CustomCoordinateHandler` accepts an axis unit specification. This extends the existing axis ordering argument to include axis units. From the perspective of the user, this would look like this::

```
ds = yt.load_uniform_grid(data, [30, 30, 30],
    bbox=np.array([[0.0, 10.0], [0.0, 30.0], [0.4, 0.9]]),
    geometry = ('custom', (('length', 'm'), ('mass', 'g'), ('time', 's')))
)
```

In this function call, note that the `geometry` argument has been extended to include both the axis ordering *and* the units that each takes. The first axis is called `length` with units of `m`, the second is called `mass` with units of `g` and the third is `time` with units of `s`.

Note that these could all be length units, but with different names – this would also be a custom coordinate system where the naming scheme can be modified.

All coordinate handlers now have an `axes_unit` dict, which maps the axis names to units.

Future developments may include allowing for specification of non-Euclidean distance functions.

27.4.5 Impact on Plotting

`PlotWindow` as a whole is designed to be used for plotting spatial datasets. Integrating non-spatial datasets presents us with two options:

- Modify `PlotWindow` such that it is generic with respect to units and aspect ratios and usable for non-spatial data.
- Utilize something like `PhasePlot` or `ParticlePlot` for plotting image data from non-spatial datasets.

At present, extremely basic plotting functionality has been put into `PlotWindow` to deal with non-spatial datasets, but this has also caused some minor impedance mismatches.

The current long-term strategy is to refactor the two plotting interfaces to share a common base class (also likely with `ParticlePlot`), and then have these choose the appropriate subclass for plotting non-spatial data and "do the right thing."

27.4.6 Future: More than Three Dimensions

Utilizing `IndexArray` is the first step toward enabling additional dimensions of data access. However, this set of functionality alone is by far insufficient. In order to enable access to greater dimensionality of data, there must be concerted effort to eliminate assumptions of 3 dimensions and generalize data structures. While this is now feasible, it is still quite the undertaking.

27.5 Backwards Compatibility

The biggest potential source of problems with backwards compatibility arise from the utilization of `YTArray` objects where `IndexArray` objects are required. This is mostly likely to happen places like centers specified to objects. However, in updating the tests, it seems that these are minimally invasive and should have only very minor impact on user-facing scripts and APIs.

Work is in progress to ensure that an `IndexArray` with homogeneous units behaves the same as a `YTArray` with those same units. This should minimize impact.

CHAPTER 28

YTEP-0028: Alternative Unit Systems

28.1 Abstract

Created: December 8, 2015 Author: John ZuHone, Nathan Goldbaum, Matthew Turk

This YTEP outlines a plan to support alternative unit systems for yt.

28.2 Status

In Progress

28.3 Project Management Links

PR: https://bitbucket.org/yt_analysis/yt/pull-requests/1904/wip-switching-between-different-base-units/

28.4 Detailed Description

28.4.1 Background

Currently, yt works with a "cgs"-based unit system. That is, all units can be expressible in a set of "base" units, which are reducible to the "centimeter-gram-second" system of units. There is one exception to this rule, that of SI current units which are not reducible to anything within the cgs system, and have a base unit of Amperes.

In the current state of the code, there is minimal support for other unit systems. The extent of this support are the methods for converting unitful quantities (`YTArrays`, `YTQuantities`) and units themselves to the SI or "MKS" (meter-kilogram-second) system. These are:

- `in_mks`: Takes a `YTArray` or `YTQuantity` and returns a new one in equivalent MKS base units.

- `convert_to_mks`: Converts the units of a `YTArray` or `YTQuantity` into the equivalent MKS base units.
- `get_mks_equivalent`: Takes a `Unit` object and returns the equivalent MKS units.

These methods are useful, but they require the user to convert to MKS "by hand" from the default "cgs" unit system used by yt, within which all calculations are carried out. Some users would prefer to work within the MKS system (or another alternative unit system) which is more appropriate for their datasets and calculations.

This YTEP outlines a proposal for allowing different unit systems to be used in yt. The core of the proposal is to allow this functionality on a per-object basis: namely, changing the unit system at the level of individual datasets, units, and unitful quantities, instead of on a global scale. The advantages of this approach are that it is relatively simple, is easily extendable, and makes only a fairly small number of changes to the fundamental code base.

28.4.2 The `UnitSystem` Object

Managing different unit systems requires the creation of a new `UnitSystem` class. A given `UnitSystem` object will consist of dict-like access to setting and getting default units with the keys corresponding to dimensions, whether strings (e.g., `"velocity"`) or SymPy `Symbol` objects registered in `yt.units.dimensions` (e.g., `yt.units.dimensions.current_mks`). Initialization of a `UnitSystem` object requires setting the name of the system, as well as a set of base units:

```
cgs_unit_system = UnitSystem("cgs", "cm", "g", "s")
```

This will initialize the `UnitSystem` along with a set of base units. The required arguments are, in order:

- `name`: The shorthand name for the `UnitSystem`.
- `length_unit`: The base length unit for this system.
- `mass_unit`: The base mass unit for this system.
- `time_unit`: The base time unit for this system.

The optional arguments are:

- `temperature_unit`: The base temperature unit for this system. Defaults to `"K"` (Kelvin).
- `angle_unit`: The base angular unit for this system. Defaults to `"rad"` (radians).
- `current_mks`: The base angular unit for current in an MKS-like system. Defaults to `None`.

If need be, the base units for temperature, angle, and MKS current can be supplied:

```
mks_unit_system = UnitSystem("mks", "m", "kg", "s",
                             temperature_unit="K",
                             angle_unit="radian",
                             current_mks_unit="A")
```

The initialization of the `UnitSystem` will also add it to a `unit_system_registry` dictionary which may be queried for a given system by its name:

```
from yt import unit_system_registry
imperial_unit_system = unit_system_registry["imperial"]
```

Once the `UnitSystem` exists, new unit defintions for specific dimensions may be added in two ways. The first is to explicitly set a unit for a specific dimension:

```
from yt.units.import dimensions
mks_unit_system["pressure"] = "Pa"
mks_unit_system[dimensions.energy] = "J"
```

So, whenever yt asks for the unit corresponding to a given dimensionality (such as in a field definition), the unit specified here will be returned. The second way to add new units to the system is simply by querying for the units for a particular dimension, without having set them previously. The effect of this is to set the units for that specific dimension by deriving them from the base units:

```
print(mks_unit_system["angular_momentum"]) # We haven't set a unit for this yet!
```

which will return `kg*m**2/s` because it will be derived from the base units of `m`, `kg`, and `s`.

Several unit systems will already be supplied for use with yt. They will be:

- `"cgs"`: Centimeters-grams-seconds unit system, with base of `(cm, g, s, K, radian)`. Uses the Gaussian normalization for electromagnetic units.
- `"mks"`: Meters-kilograms-seconds unit system, with base of `(m, kg, s, K, radian, A)`.
- `"imperial"`: Imperial unit system, with base of `(mile, lbm, s, R, radian)`.
- `"galactic"`: "Galactic" unit system, with base of `(kpc, Msun, Myr, K, radian)`.
- `"solar"`: "Solar" unit system, with base of `(AU, Mearth, yr, K, radian)`.
- `"planck"`: Planck natural units ($\hbar = c = G = k_B = 1$), with base of `(l_pl, m_pl, t_pl, T_pl, radian)`.
- `"geometrized"`: Geometrized natural units ($c = G = 1$), with base of `(l_geom, m_geom, t_geom, K, radian)`.

Users may create new `UnitSystem` objects on the fly, which will be added to the `unit_system_registry` automatically as they are created. Both of these will be accessible from the top-level `yt` module.

"code" UnitSystems

When a dataset is instantiated, a `UnitSystem` object corresponding to the code units for that dataset will be created and added to the `unit_system_registry`, where the name will be the string representation of the `Dataset` object:

```
from yt import unit_system_registry, load
ds = load("GasSloshing/sloshing_nomag2_hdf5_plt_cnt_0100")
sloshing_unit_system = unit_system_registry[str(ds)]
```

28.4.3 Unit Systems and `Dataset` objects

The main user-facing interface to the different unit systems will be through the `load` function. `load` will take a new keyword argument, `unit_system`, which will be a string that corresponds to the name identifier for the desired unit system, with a default value of `"cgs"`. The main effect of changing the unit system will be to return all aliased fields and derived fields in the units of the chosen system. For example, to change the units to MKS in a FLASH dataset:

```
ds = yt.load("GasSloshing/sloshing_nomag2_hdf5_plt_cnt_0100", unit_system="mks")
sp = ds.sphere("c", (100.,"kpc"))
print(sp["density"])
print(sp["flash","dens"])
print(sp["kinetic_energy"])
print(sp["angular_momentum_x"])
```

```
[  1.30865584e-23   1.28922012e-23   1.30364287e-23 ...,   1.61943869e-23
   1.61525279e-23   1.59566008e-23] kg/m**3

[  1.30865584e-26   1.28922012e-26   1.30364287e-26 ...,   1.61943869e-26
   1.61525279e-26   1.59566008e-26] code_mass/code_length**3

[  6.37117204e-13   6.12785535e-13   6.20621019e-13 ...,   3.12205509e-13
   3.01537806e-13   3.39879277e-13] Pa

[ -3.97578436e+63  -3.92971077e+63  -3.95375204e+63 ...,   2.39040654e+63
   2.39880417e+63   2.44245756e+63] kg*m**2/s
```

Note that in this example, `"density"` is an alias to the FLASH field `("flash","dens")`, and it has had its units converted to MKS, but the original FLASH field remains in its default code units. `"kinetic_energy"` and `"angular_momentum_x"` are derived fields which have also had their units converted.

Another option is to express everything in terms of code units, which may be achieved by setting `unit_system="code"`:

```
ds = yt.load("IsolatedGalaxy/galaxy0030/galaxy0030", unit_system="code")
sp = ds.sphere("c", (30., "kpc"))
print(sp["density"])
print(sp["kinetic_energy"])
print(sp["angular_momentum_x"])
```

```
[    744.93731689     717.57232666     682.97546387 ...,   40881.68359375
   57788.68359375  397754.90625   ] code_mass/code_length**3

[    97150.95501972     91893.64007627     85923.44662925 ...,
  11686694.21560157  16358988.90006877  79837013.8427877 ] code_mass/(code_
↪length*code_time**2)

[ -1.17917130e-10  -1.05648103e-10  -9.26664470e-11 ...,   2.05149702e-09
   2.03607319e-09   6.72304619e-09] code_length**2*code_mass/code_time
```

Currently, the plan is to have all frontends allow the user to set `unit_system` in the call to `load`, but this should be evaluated on a per-frontend basis. For some frontends, it may be more appropriate to set the unit system explicitly, whether to `"cgs"` or some other system.

28.4.4 Using `UnitSystems` in Field Definitions

In order for derived fields to take advantage of the different unit systems, it will be necessary to change the units in the field definitions, so that the derived fields may be returned in the units of the system specified when the dataset was loaded.

For example, in setting up the specific angular momentum fields in `yt.fields.specific_angular_momentum`, we would change the units thus:

```
def setup_angular_momentum(registry, ftype = "gas", slice_info = None):
    unit_system = registry.ds.unit_system
    def _specific_angular_momentum_x(field, data):
        xv, yv, zv = obtain_velocities(data, ftype)
        rv = obtain_rvec(data)
        rv = np.rollaxis(rv, 0, len(rv.shape))
        rv = data.ds.arr(rv, input_units = data["index", "x"].units)
        return yv * rv[...,2] - zv * rv[...,1]
```

(continues on next page)

(continued from previous page)

```
    ...

    registry.add_field((ftype, "specific_angular_momentum_x"),
                       function=_specific_angular_momentum_x,
                       units=unit_system["specific_angular_momentum"],
                       validators=[ValidateParameter("center")])
```

Notice that the field definition code itself has not been altered at all except that the `units` keyword argument to `registry.add_field` has been changed from `cm**2/s` to `unit_system["specific_angular_momentum"]`, which will set the units for the field to whatever is appropriate for the unit system associated with the dataset. The `unit_system` object may be queried with either SymPy `symbol` objects in `yt.units.dimensions` or strings corresponding to the variable names of those objects.

This will not be appropriate for all fields–some fields naturally belong in certain units regardless of the underlying system used. In the context of galaxy clusters, `"entropy"` is an example, which naturally belongs in units of `keV*cm**2`. Whether or not to change units should be evaluated on a per-field basis.

For users adding their own derived fields, there will be two ways to take advantage of the new unit systems functionality. If derived fields are being created from a dataset using `ds.add_field`, they can set up the units in a similar way as above:

```
def _density_squared(field, data):
    return data["density"]*data["density"]
ds.add_field(("gas","density_squared"), function=_density_squared, units=ds.unit_
↪system["density"]**2)
```

If using `yt.add_field`, however, it will be necessary to set `units="auto"` in the call to `add_field`. To provide an extra layer of error handling for this case, a `dimensions` keyword argument will be added to the `DerivedField` initialization, which will only be used if `units="auto"`, and will be used to check that the dimensions supplied to `add_field` and the dimensions of the `YTArray` in the field definition are the same:

```
from yt.units.dimensions import temperature

inverse_temp = 1/temperature

def _inv_temperature(field, data):
    return 1.0/data["temperature"]
yt.add_field(("gas","inv_temperature"), function=_inv_temperature, units="auto",
             dimensions=inverse_temp)
```

If one does not supply a `dimensions` argument when `units="auto"`, or if the dimensions are incompatible, errors will be thrown.

Special Handling for Magnetic Fields

Making magnetic fields compatible with different unit systems requires special handling. The reason for this is that the units for the magnetic field in the cgs and MKS systems are not reducible to one another. Superficially, it would appear that they are, since the units of the magnetic field in the cgs and MKS system are gauss (G) and tesla (T), respectively, and numerically $1\ \mathrm{G} = 10^{-4}\ \mathrm{T}$. However, if we examine the base units, we find that they have different

dimensions:

$$1\ \mathrm{G} = 1\ \frac{\sqrt{\mathrm{g}}}{\sqrt{\mathrm{cm}}\cdot\mathrm{s}}$$
$$1\ \mathrm{T} = 1\ \frac{\mathrm{kg}}{\mathrm{A}\cdot\mathrm{s}^2}$$

It is easier to see the difference between the dimensionality of the magnetic field in the two systems in terms of the definition of the magnetic pressure:

$$p_B = \frac{B^2}{8\pi}\ (\mathrm{cgs})$$
$$p_B = \frac{B^2}{2\mu_0}\ (\mathrm{MKS})$$

where $\mu_0 = 4\pi \times 10^{-7}\ \mathrm{N/A^2}$ is the vacuum permeability. Therefore, in order to handle the different cases of the magnetic field units for the two different systems, it is necessary to have field definitions which can take the different dimensionalities into account.

The most fundamental change which is required will be to change the way aliases are handled for the magnetic field vector fields. Normally, the dataset field and the aliased field will have the same dimensions. For example, in the case of a FLASH dataset, `("flash","magx")` and its alias `("gas","magnetic_field_x")` will have the same dimensions of `magnetic_field_cgs`, which are `sqrt((mass))/(sqrt((length))*(time))`. This is handled by specifying the alias in the `known_other_fields` atttribute of the `FieldInfoContainer` like this:

```
class FLASHFieldInfo(FieldInfoContainer):
    known_other_fields = (
        ...
        ("magx", (b_units, ["magnetic_field_x"], "B_x")),
        ("magy", (b_units, ["magnetic_field_y"], "B_y")),
        ("magz", (b_units, ["magnetic_field_z"], "B_z")),
        ...
    )
```

Where the alias is the second item in the 3-element tuple after the field name. However, we may want to convert from a cgs unit system to an MKS unit system, which would require changing the dimensions of the alias `"magnetic_field_x"` (while leaving the units and dimensions of the dataset field `"magx"` intact). The solution is to remove the alias from `known_other_fields` and supply a helper function which creates the aliases, taking into account the specified unit system:

```
class FLASHFieldInfo(FieldInfoContainer):
    known_other_fields = (
        ...
        ("magx", (b_units, [], "B_x")), # Note the alias has been removed
        ("magy", (b_units, [], "B_y")),
        ("magz", (b_units, [], "B_z")),
        ...
    )

    def setup_fluid_fields(self):
        from yt.fields.magnetic_field import \
            setup_magnetic_field_aliases
        ...
        setup_magnetic_field_aliases(self, "flash", ["mag%s" % ax for ax in "xyz"])
```

Again, this will have to be evaluated on a per-frontend basis as to what is most appropriate for the handling of the magnetic field units. The definitions for other magnetic-related fields such as `"magnetic_pressure"` and `"alfven_speed"` will also be modified to ensure that the units are handled properly for the different systems.

28.4.5 Other Ways to Use the Unit Systems

There will be other ways in which unit-aware objects in yt may be converted to a different unit system. they are:

`in_base`, `convert_to_base`, `get_base_equivalent` methods

These three methods, which currently convert unitful quantities and units to the yt base units of cgs (plus Ampere if the dimensionality includes `current_mks`), will be modified to include a `unit_system` keyword argument, which will be set to `"cgs"` by default. The purpose of this keyword argument is to allow switching between different unit systems for `YTArrays`, `YTQuantities`, and `Unit` objects. This keyword argument may be set to a string corresponding to the name of the desired unit system. Some examples:

```
a = YTArray([1.0, 2.0, 3.0], "km/hr")
print(a.in_base("imperial"))
```

```
[ 0.91134442,  1.82268883,  2.73403325] ft/s
```

```
b = YTQuantity(12., "g/cm**3")
b.convert_to_base("galactic")
print(b)
```

```
1.7730691071344677e+32 Msun/kpc**3
```

```
c = YTQuantity(100., "mile/hr")
print(c.units.get_base_equivalent("mks"))
```

```
m/s
```

Alternatively, a `Dataset` object may be passed as the `unit_system` argument, which will convert to the base code units of that dataset:

```
ds = yt.load("IsolatedGalaxy/galaxy0030/galaxy0030")
sp = ds.sphere("c", (30., "kpc"))
print(sp["density"].in_base(ds))
```

```
[    744.93731689     717.57232666     682.97546387 ...,   40881.68359375
   57788.68359375  397754.90625   ] code_mass/code_length**3
```

Note that this will only work if the `YTArray`, `YTQuantity`, or `Unit` in question "knows" about those code units, e.g., it is from a data container from that `Dataset` or was initialized using `ds.arr`.

A call to `in_base` or `convert_to_base` without specifying a unit system will convert to the default "cgs" unit system:

```
a = YTArray([1.0, 2.0, 3.0], "km/hr")
print(a.in_base())
```

```
[ 27.77777778,  55.55555556,  83.33333333] cm/s
```

which is the current behavior of the code, ensuring backwards-compatibility. The behavior of the cgs and MKS-specific methods (e.g., `in_cgs`, `in_mks`, etc.) will not be modified.

`Cosmology` object

Currently, the `Cosmology` object returns all quantities in cgs units. The proposed changes will add a new keyword argument, `unit_system`, which will be a string that corresponds to the name identifier for the desired unit system, with a default value of `"cgs"`.

```
cosmo = Cosmology(unit_system="galactic")
```

Alternatively, `unit_system` may be set to a `Dataset` object to use the code units of that dataset:

```
ds = yt.load("IsolatedGalaxy/galaxy0030/galaxy0030")
cosmo = Cosmology(unit_system=ds)
```

28.4.6 Physical Constants in the Different Unit Systems

Each `UnitSystem` object will have a `constants` attribute which can be used to obtain any physical constant in `yt.utilities.physical_constants` in the base units of that system. For example:

```
import yt

galactic_units_system = yt.unit_system_registry["galactic"]

G = galactic_units_system.constants.G
clight = galactic_units_system.constants.clight
mp = galactic_units.system.constants.mp

print(G)
print(clight)
print(mp)
```

```
4.498205661165364e-12 kpc**3/(Msun*Myr**2)

306.6013938381177 kpc/Myr

8.417430465502256e-58 Msun
```

28.4.7 Notifying the Community

The community will be notified about this feature enhancement via the mailing list and appropriate social media accounts. Appropriate documentation of this feature will be added.

28.5 Backwards Compatibility

Since the base unit system for all yt units will remain cgs, and the `unit_system` keyword will always default to `"cgs"` for loading datasets, setting up `Cosmology` objects, and unit conversions of arrays, the changes as proposed are fully backwards-compatible.

28.6 Alternatives

The only alternative discussed up to this point was to set the unit system globally for a given yt session using the configuration system. The system proposed here allows for more fine-grained control at the level of individual objects, e.g. `Dataset`, `YTArray`, and `Cosmology` objects, which should be sufficient for most (if not all) purposes. Another option is to make the default base units themselves configurable. This is disfavored since it does not appear to add additional functionality beyond the currently proposed scheme, and would result in more widespread changes to the code base.

CHAPTER 29

YTEP-0029: Extension Packages

29.1 Abstract

Created: January 25, 2016

Author: Matthew Turk

The yt project is not the same as the yt codebase. However, the bundling of analysis modules and maintaining a monolithic repository tends to blur the distinction between the two. This YTEP is an attempt to identify how we can make the codebase more friendly to extension modules, and to encourage an ecosystem of independently developed packages that utilize yt.

29.2 Status

Completed

29.3 Project Management Links

There has been discussion on yt-dev about this, but due to Dreamhost's constant problems with archiving mail, they may be gone forever, about which I will refrain from editorializing here.

29.4 Detailed Description

This YTEP proposes a few courses of action designed to promote the idea of yt as a dependency, rather than a destination, for analysis modules and external projects.

As it stands, many (the author included) have viewed yt as the "place" to put things, whether or not they contribute to its core mission. There are several good reasons for this:

- **Distribution**: yt is largely a monolithic codebase, and if you have downloaded and installed yt, you have access to all of the analysis modules. This makes discoverability and distribution much easier.
- **Infrastructure**: yt has testing infrastructure which gets run on all pull requests and new revisions. As such, if the analysis module is in the primary codebase, it too will be tested. If functionality in yt changes, the analysis module will track this. Additionally, we provide the ability to run tests on managed infrastructure.
- **Hosting**: We will allow these projects to host data on hub.yt as well as have their websites either as subrepositories in the main *yt-project/website* repository or as subdirectories. This will allow them to share our hosting infrastructure without developing their own, if they so wish.
- **Prestige**: (maybe?) Having something be a part of yt adds to the sense that it's part of the bigger project, and that it's somehow graduated to a stable component. (I am not sure this is real, but it's been mentioned to me.)
- **Social infrastructure**: There's a mailing list for yt, so folks know where to go.

But, there are several – sometimes quite large – downsides to being part of the mainline yt codebase.

- **Project Standards**: The methods right now for developing in yt require several iterations of code review as well as conforming to standards of development practices. For instance, this could include style or where the code is developed (i.e., bitbucket/github/kallithea/etc).
- **Credit**: While the situation is certainly improving, and there is nothing to stop a contributor from writing a paper describing a new analysis module (in fact this is encouraged) it is still a part of a "project" codebase rather than a standalone entity. This can diminish the perception of individual contributions, particularly by people not directly affiliated with the mainline yt project.
- **Timeline**: The acceptance of a change into yt can often be on a weeks or longer timescale, depending largely on the time being issuing a pull request and the next PR triage. For modules that are developed with much higher frequency, this can be very cumbersome and costly in developer efforts.
- **Dependencies**: yt attempts to keep the number of dependencies at a minimum; if a package wants to use them, it often has to either make them on-demand or vendor them with the source. We want to encourage packages to utilize new technologies and experiment – but at present, we can't allow that into mainline.

A solution to this problem would be to encourage an ecosystem of discoverable packages that build on yt as a dependency. Three such packages exist at time of writing – powderday, trident, and astroblend.

What this YTEP proposes is to restructure our discussion of yt as a project to emphasize these types of packages and projects as citizens in the community, and provide to them a mechanism for people to find them.

This will take a few forms:

- Re-designing the website to emphasize the burgeoning ecosystem of projects, perhaps similar to the way astropy.org is set up.
- Offering, but not mandating, projects that they should direct questions to the yt-users mailing list.
- Splitting out some particularly isolated projects from analysis modules. This may include photon simulator and the sz generator.
- Provide entry points *in the code* for people to find out about available analysis modules. (Discussed below.)
- Utilize intersphinx documentation links for analysis modules, *or* offer to host documentation on the main yt homepage.
- Determine mechanisms for recognizing projects as extensions, independent, related, etc.
- Adding testing support can be difficult and labor intensive; however, we should explore adding testing support for extensions under some criteria. (Such as, does this depend on public APIs, do test failures constitute upstream breakages, etc.)
- Accept either subrepos or subdirectories in *yt-project/website*.

At present, people can find analysis modules by doing something like::

```
>>> dir(yt.amods)
```

This returns a list of analysis modules. This could be extended to *also* parse a small list of known external projects which could be installed. Importing one that was not installed would report that it wasn't installed and describe a method for installing it.

Depending on the type of project, we may want to distinguish between yt extensions and projects that share the yt ecosystem. For the former, we could evaluate using `yt.extension` as the namespace, if `yt.analysis_module` is not appropriate. This would then provide (similar to how Flask manages extension modules) a naming scheme (say, `ytext_projectname`) that would be imported into `yt.extension`. This is not necessary, but is a possibility within the scope of this YTEP.

29.5 Backwards Compatibility

If we split out existing modules, they would need to be API compatible. We would also want to make sure that they are well-supported by our infrastructure.

29.6 Alternatives

One possible alternative would be to make the `yt.analysis_modules` namespace a free-for-all of modifications, with much shorter timescales and essentially autonomous operation by developers.

CHAPTER 30

YTEP-0031: Unstructured Mesh

30.1 Abstract

Created: December 18, 2014

Author: Matthew Turk

Supporting data generated by unstructured mesh codes can be implemented using a combination of existing data selection routines, vector fields, index fields, and pixelization routines. This YTEP also touches on decoupling the step of constructing an image buffer from data and the data representation itself, which applies to data types such as SPH as well as mixing different mesh types in a single `Dataset` object.

30.2 Status

Proposed

30.3 Project Management Links

Any external links to:

- WIP pull request for hexahedral mesh: https://bitbucket.org/yt_analysis/yt/pull-request/1370/wip-generic-hexahedral-mesh-pixelizer/diff

Some small discussion has occurred on-list as well, but not yet of too much detail.

30.4 Detailed Description

30.4.1 Background

Data in yt has until now been assumed to be either discrete, or on a regular mesh that is defined by cells. This has been the implementation for smoothing kernels as well as particle and particle depositions. However, to support broader codes (and to support more flexible visualizations of particle datasets, such as direct smoothing kernel application onto image buffers) we should develop methods for supporting unstructured mesh. This will require uncoupling the pixelization step not only from coordinates (as has been done) but also from the underlying data model. This YTEP proposes a strategy for implementing support for unstructed meshes. We will not restrict ourselves to specific element types; however, where particular geometries are required, we will first focus on hexahedral, tetrahedral, wedge, pyramid and voronoi tesselation cells. The YTEP as a whole is designed to be generic with respect to element geometry.

We will not spend much time discussing Voronoi tesselation cells here and will instead reserve discussion of them for a future YTEP, as they are similar in some ways but quite different in others. Additionally, we restrict ourselves initially to *convex* elements.

In conjunction with these different cell types, data may be stored at vertices, centers, edges, faces, and so on. Developing the appropriate intra-cell interpolation methods will remove the need to explicitly identify these different data "centering" methods, but see below for discussion of how these centering methods will be denoted.

Typically within an element, the value at a given position is a function of a kernel that uses as input the quantities defined in the output. So given the kernel, and the data points, the value at any internal position in the cell can be determined. Supporting these interpolation kernels is part of a longer-term roadmap. Initially, we will likely perform nearest-neighbor or IDW.

The difficulty in handling unstructed mesh data is ensuring high-fidelity visualizations as well as high-fidelity data representation. The latter is considerably more straightforward, as if the system is viewed as a sequence of discrete points (and the question of intra-element interpolation is ignored except when *explicitly requested*) the existing selection routines can be used by regarding the data as a sequence of discrete points ("particles").

We assume for the purposes of this document that the datasets are defined by connectivity and coordinates (i.e., an array of indices that define each element's vertices and an array of common coordinates into which we index) and that the ordering of the vertices is such that we can identify which vectors span faces of the elements.

30.4.2 Irregular Mesh Points

In *YTEP-0001: IO Chunking* a system for chunking was described, wherein attributes such as *fcoords*, *icoords* and so on are exposed. We propose to augment these, which work nicely for regular mesh data, with coordinates for vertices and edges. However, this will only be accessible for chunks which are homogeneous in element type. (Otherwise, the vector would need to be "ragged" with differing component length.)

What this YTEP proposes to do is to create a new set of fields and chunk attributes. The fields, much like fields such as `x`, `y` and `z`, will reflect the position of vertices and/or faces of an element and will be vector fields. The new chunk attributes will be the underlying data from which these fields are generated. The new chunk system will add on these attributes to `YTDataChunk`:

- `fcoords_vertex` - vertex positions, of shape (N, Nvertex, 3)
- `fcoords_face` - barycenter of the faces, of shape (N, Nface, 3)
- `fcoords_edge` - middle of edge, of shape (N, Nedge, 3)

We anticipate that for the existing data index types, these attributes can be created on the fly, but will not often be used.

The new fields will follow the naming convention of `element_type, type_coordinateaxis`. A few examples:

- `("hex", "vertex_x")` - vector of shape (N, Nvertex)
- `("hex", "face_x")` - vector of shape (N, Nface)
- `("tetra", "face_phi")` - vector of shape (N, Nface)
- `("wedge", "edge_z")` - vector of shape (N, Nedge)
- `("vertex", "x")` - array of shape (N)
- `("edge", "x")` - array of shape (N)

We will still retain existing `index` fields, which will return centroids for coordinates and will simply return invalid coordinate requests for the path element and cell width fields, similar to when fields like `dx` are requested in non-Cartesian coordinates. This should work, although it's not entirely clear that it is the best bet. Namespacing could potentially fix this.

What this will provide:

- Uniform access to coordinates of all raw, unprocessed data points.
- Re-use of existing routines that query based on field type and field name.
- Avoid confusion based on re-use of names from other systems of coordinates.

Likely this will also need a method for transforming values between definition systems; for instance, a method for converting vertex-centered values to cell-centered. This would be akin to the method used for depositing particles on a mesh and would mandate access to the mesh object via `ValidateSpatial`.

30.4.3 Decoupling Pixelization from Mesh Values

The pixelization step is the point at which mesh values are transformed into an image. These mesh values are variable resolution, and so the operation essentially deposits (through NN interpolation with anti-aliasing) these variable mesh values into an image buffer.

In cases where the mesh values are accessible through the fields used currently (such as `px` and the like), the standard pixelization routines will be called.

For datasets that do not, or cannot, create `px` fields and the like, separate pixelization routines will be called. In the (at time of writing) WIP PR for hexahedral mesh datasets, and example of this can be found. This will be implemented in the coordinate handler.

The generic pixelization routine will accept a set of vertices, an interpolation kernel (nearest-neighbor for starters) and the field (initially only support for fields defined at centroids will be added for simplicity, but with edge and face added later). The ordering of vertices that provides face values will be specified at pixelization time, and will draw from one of a set of orders.

The pixelization routine will first apply coarse bounding box checks to the image plane and all supplied elements. Each pixel that passes the bounding box check for a given element will move on to the second step of selection. In this step, the sign of the dot product of the centroid with each normal vector defining each face will be stored (this prevents the need for knowing the CW / CCW ordering of the vertices) and for each pixel in the image plane, the signs of the same dot product will be examined. If all the signs match, the point is internal to our (convex) element. This appropriate kernel will be evaluated and the resulting value deposited in the image plane.

Because of the requirements of single mesh type, the pixelization routines will iterate over each mesh type and deposit the fields in sequence. This will enable the interoperation of fields between mesh types, without requiring that they be made uniform in size.

Note also that separating out based on the type of field and data represented means that we may now be able to implement slices of particle fields directly.

30.4.4 Multiple Meshes for Multiple Mesh Types

Each mesh type – hex, tet, wedge, etc – will be isolated to a different mesh type.

For a given data object, much like particles and mesh objects cannot interact without the mediation of a deposition step, each must be queried separately if the vertices are to be examined. If the field values are the only items of concern, they can be queried in concatenated form. For situations where fields persist across mesh types, we will be unable to supply vertex information and can only then supply `x` fields and the like.

At present, there is a semi-structured mesh object, and for datasets that expose that, it lives within the `.meshes` attribute of the index. Each mesh type will be in a separate element in that list.

30.4.5 Example Use Cases

These example use cases should *just work* in a successful implementation. The dataset imagined in them contains tetrahedra (`N_t`), hexahedra (`N_h`), and wedges (`N_w`). The field `field1` is defined at vertices and `field2` is defined at the element centroids.

Querying all of the values of `field1`:

```
dd = ds.all_data()
print dd["vertex", "x"].shape
print dd["index", "x"].shape
print dd["field1"].shape
```

The first and third print statements will return the same shape, but the middle will return the total number of elements (centroids). Ultimately, much like with particle fields, the user will need to have some knowledge of the mesh (which yt can provide hints about) to know how to combine fields.

This should also work:

```
prof1d = yt.create_profile(dd, ("vertex", "x"), "field1")
```

Because our selection operators will operate on the field values as though they were discrete points, this must also work:

```
sp = ds.sphere([0.5, 1.0, 30.1], (1.0, "km"))
sp["field1"]
sp["field2"]
```

These fields will not be the same size, but will select from all different mesh types. Querying the `"x"` field will return the centroids that pass the selector, which will be of different size than `"field1"` but will be the same size as `"field2"`. This also means that it will be impossible to bin `"field1"` against `"x"` without explicitly namespacing it as `("vertex", "x")`.

30.4.6 Volume Rendering

Initial support for volume rendering will use Embree, a fast ray-tracing code from Intel, to do the ray traversal. A set of python bindings for Embree already exists. Later on, this may be replaced our own ray-tracing code to remove the external dependency.

To use Embree, we must write code that generates a Triangular polygon mesh from the unstructured mesh data yt reads in. This may involve breaking up faces into multiple triangles. Currently, this is implemented for Hexahedral and Tetrahedral mesh elements, and adding support for other mesh types should not be difficult. One then uses the functions Embree provides to cast rays at the resulting mesh.

There will be two basic "plot types" for volume renderings of unstructured mesh data. The first will be "surface plots", where the value of the field at the intersection point with each ray will be calculated using hit data computed by Embree. The second will be more like the traditional yt volume renderings, values along each ray will be accumulated for every element the rays intersect. For example, one could compute the maximum intensity along each ray instead of the value on the surface. Both of these types of renderings will need implementations of various intra-element interpolation functions to support meshes of various types and orders.

All of this will be integrated in with the Volume Rendering refactor, so that we retain the flexibility provided there for creating movies and camera paths. This will involve (at least) defining a new type of RenderSource object for polygon meshes. This object will know how to create the Embree polygon mesh from the data_source that gets passes in, and how to do the appropriate ray tracing calls. Once this source has been created, the Camera will be able to be changed at will, as defined in the YTEP for the scene refactor. Because multiple RenderSource objects can exist in the same scene, there is no reason why different meshes with different plot types can't exist in the same scene.

Some examples of what the volume renderings will look like are here: https://www.dropbox.com/s/xx2it8p0ivk7s69/surface_render_0.png?dl=0 https://www.dropbox.com/s/m0b9wdp6uh6h4nm/surface_render_1.png?dl=0

30.4.7 Explicitly Not Implemented Functionality

These pieces of functionality will need considerable reworking before they will be suitable for use with unstructured mesh data, and they are outside of the scope of this document:

- "Spatial" fields, as connectivity between elements is not well-defined in general (although it may be for specific element types)
- Block and tile iterators, as they are not immediately relevant to unstructured meshes

These are difficult, and we will be holding off on implementing them until this YTEP and its implementation have shaken out.

30.5 Backwards Compatibility

This should have absolutely no backwards incompatible changes; any backwards-incompatible changes will be considered bugs and will result in a redesign.

30.6 Alternatives

A few alternatives exist. For instance, instead of augmenting `fcoords` and so on with new definitions, we could either define new fields and leave `fcoords` to refer to centroids (or delete it for those objects), or we could define vector fields for these that are of shape (N, Ncell, 3), and refer to the vertices of the data.

Additionally, we could be more explicit about what refers to what; we could have different namespaces for vertices.

Another alternate idea would be to mimic the particle method for namespacing and positions; this would result in things like `("field_type", "hex_vertex_x")` and so on. Or, we could do `("hex_vertex", "x")` and similar.

30.6.1 Open Questions

- Should we get rid of `particle_type` and replace with a classification such as `centroids`, `discrete`, `vertex` and so on?

- How should we handle namespacing for fields that may be defined at multiple places (face *and* vertex, for instance)

CHAPTER 31

YTEP-0032: Removing the global octree mesh for particle data

31.1 Abstract

Created: February 9 2017
Author: Nathan Goldbaum, Meagan Lang, Matthew Turk

The global particle octree index used by yt presents a barrier for improving the performance and scalability of visualizing and analyzing particle datasets. This YTEP proposes removing the global octree index, replacing it with a combination of a new IO system and changes to the high-level yt API to focus on returning particle-centric data. The particle I/O refactor makes use of an indexing scheme based on compressed Morton bitmaps which dramatically improves memory usage and scaling for large particle datasets by eliminating the need for a global octree index.

Rather than constructing a global index to maintain backward compatibility at the cost of scaling and performance, we instead propose a reworking of the yt user interface for particle and SPH data to be more "particle-centric". This means that data object selections for fields that are now defined on the global octree mesh will instead return field data at particle locations. For SPH data, visualizations of slices and projections are done in the image plane, making use of the "scatter" approach by smoothing SPH data directly onto images, employing either a volumetric or projected SPH smoothing kernel. Fully local derived fields are calculated using yt's existing field definitions but passing in data defined at particle locations. Fields that need spatial derivatives are implemented using the SPH formalism and are also evaluated at the particle locations.

Altogether these changes allow for improved performance and scaling, and allow users to access, analyze, and visualize particle field data for SPH simulations in a more straightforward fashion. While we do not propose substantial API changes for mesh or octree codes, these changes to yt's field system for particle data imply substantial changes to the *meaning* of yt's data selection system for particle data. We discuss the implications of these backward incompatible changes and how we intend to document and manage them in a way that is minimally disruptive to users.

31.2 Status

In Progress. The implementation is mostly finished, although there are a few features that still need to be implemented.

31.3 Project Management Links

The code can be found in pull request 2382:

https://bitbucket.org/yt_analysis/yt/pull-requests/2382

The C++ compressed bitmap implementation we intend to vendor into yt:

https://github.com/lemire/EWAHBoolArray

31.4 Detailed Description

31.4.1 Background

Currently most user-facing operations on SPH data are produced by interpolating SPH data onto a volume-filling octree mesh. When support for SPH data was added to yt in the run-up to the yt-3.0 release, this approach allowed yt to support SPH data in a way that could reuse the existing infrastructure in yt for octree data and preserve core assumptions in yt that `gas` fields must correspond to a volume-filling AMR structure. While this did make initial support for SPH data much easier, it also had some downsides. In particular, because the octree was a single, global object, the memory and CPU overhead of smoothing SPH data onto the octree can be prohibitive on particle datasets produced by large simulations. Constructing the octree during the initial indexing phase also required each particle (albeit, in a 64-bit integer) to be present in memory simultaneously for a sorting operation, which was memory prohibitive. Visualizations of slices and projections produced by yt using the default `n_ref` and `over_refine_factor` are somewhat blocky since by default we use a relatively coarse octree to preserve memory. In addition, since we construct the global octree based on the positions of all particles, visualizations that include only one particle type sometimes include "holes" in regions under-sampled by that particle type.

These cosmetic and semantic issues are jarring to users of SPH codes, who tend to think of data defined at the particle locations rather than sampled onto an adaptive mesh. Making our high-level API focus more on particle-centric data will help to ease the cognitive dissonance felt by users of SPH codes when they work with yt.

Over the past two years, Meagan Lang and Matt Turk implemented a new approach for handling I/O of particle data, based on storing compressed bitmaps containing Morton indices instead of an in-memory octree. This new capability means that the global octree index is now no longer necessary to enable I/O chunking and spatial indexing of particle data in yt.

In this document we describe the approach we take for replacing the global octree index with Morton bitmasks. First, we describe the implementation of the Morton bitmask index, changes to the low-level selector API needed to support the Morton bitmask work, and the testing strategy used for the Morton bitmap indexing scheme. Next we discuss high-level changes to how yt handles particle data, changes to the field system for SPH data, the implementation of the SPH pixelizer system for visualizations, a discussion of deposit fields, and a description of the strategy used to test the new approach for SPH data. We close with a discussion of questions that still need to be tackled before this work can be merged.

31.4.2 Low-Level Implementation

Morton Bitmap Index

The generated index serves to map how the domain is populated by particles in datasets split across multiple files. This way, spatial queries can skip files that do not contain particles in the selected part of the domain. The files are mapped by storing two nested Morton indices for each particle in a dataset. Rather than storing the indices in plaintext, we make use of an EWAH compressed boolean array bitmap. Given a domain with known boundaries in each dimension, a 3-dimension position can be described by a single integer Morton index by

1. Dividing the domain into `2^index_order1` cells in each dimension with widths `ddx = domain_width_x/(2^index_order1)`.
2. Determining the 3 integers specifying the cell that contains the 3D position (e.g. `x/ddx`).
3. Combine the 3 integers into a single integer by alternating bits from each dimension.

These indices can be stored as either integers or in boolean masks. In the case of the mask, an array of zeroed bits is created with a length equal to the maximum possible index for the chosen value of `index_order1`. Then the bits for the indices present are set to one. To save space, boolean masks in the form of bitmaps can then be compressed further using the Enhanced Word-Aligned Hybrid (EWAH) bitmap compression algorithm. In practice, we make use of a vendored version of EWAHBoolArray, a C++ EWAH bitmask implementation available under the Apache v2 license.

One bitmap is created for each file. If an index is present in more than one file's bitmap, this represents a collision and decreases the likelihood that the bitmap can be used to exclude files during spatial queries. This is unlikely if the particles are well partitioned between the files according to a domain decomposition scheme at the chosen order, but this is not generally true of particle datasets produced by astrophysical simulations. In these cases, it is better to create a more refined index.

Using a larger `index_order1` increases the refinement of the index, but also increases both the memory required to store the indices and the time required to query them for EWAH bitmaps. To combat this, we include a second refined index within those cells that have indices in multiple files' bitmaps for the coarse index. For each particle with a coarse index that collides with another file, a second refined Morton index is creating by following the same procedure as for the coarse index, but exchanging the domain boundaries for the boundaries of the coarse index cell. The refined index for each file is then stored in a EWAH bitmap for each coarse cell with a collision.

The coarse and refined indices are generated in two separate I/O passes over the entire dataset. To generate the coarse index, the coordinates of all particles, as well as the softening lengths for SPH particles, are read in from each file. For each particle we then compute the Morton index corresponding to the particles position within the domain. This index, `mi` is then used to set the `mi`th element in a boolean mask for the file to 1. If the particle is an SPH particle, neighboring indices with cells that overlap a sphere with a radius equal to the particle's softening length and centered on the particle are also set to 1.

Once a coarse boolean mask is obtained for each file, the masks are stored in a set of EWAH compressed bitmaps (implemented in the `ewah_bool_array` Cython extension classes). Using logical boolean operations, we then identify those indices that are set to 1 in more than one file's mask (the collisions). The EWAH format is particularly good at logical operations, as it does not necessarily require decompression to determine whether or not particular bits are set.

During a second I/O pass over the entire dataset, refined indices are created for those particles with colliding coarse indices. Both the coarse and refined indices are stored in an array for each file. One a file has been completely read in, those indices are sorted and used to create a map from coarse indices to EWAH compressed bitmaps. This is done because entries in EWAH compressed bitmaps must be set in order.

The Morton bitmap index is created for each particle dataset upon its first ingestion into yt and saved to a sidecar file. At all future ingestions of the dataset into yt, the index will be loaded from the sidecar file. Indexes are managed through the Cython extension class `ParticleBitmap` (defined in `yt/geometry/particle_oct_container.pyx`), which is exposed to the user visible yt API via the `regions` attribute of the `ParticleIndex` class (e.g. `ds.index.regions`). The `ParticleBitmap` class generates EWAH bitmaps via the `BoolArrayCollection` Cython extension object (defined in `yt/utilities/lib/ewah_bool_wrap.pyx`), which wraps the underlying `EWAHBoolArray` C++ library.

In the current implementation users can control the creation of the bitmask index via the `index_order` and `index_filename` keyword arguments accepted by `SPHDataset` instances. These keyword arguments replace the deprecated `n_ref`, `over_refine_factor` and `index_ptype` keyword arguments. The `index_order` is a two-element tuple corresponding to the maximum Morton order for the coarse and refined index. Using a tuple for the `index_order` instead of two keyword arguments is not only more terse, but it will allow us to produce bitmask indexes in the future with multiple refined indices while maintaining the same public API. Currently the default

`index_order` is `(7, 5)`. If a user specifies `index_order` as an integer, the integer is taken as the order of the coarse index and the order of the refined index is set to `1`, producing a trivial refined index. For example:

```
import yt
ds = yt.load('snapshot_033/snap_033.0.hdf5',
             index_order=(5, 3), index_filename='my_index')
ds.index
```

Running this script will produce the following output:

```
yt : [INFO     ] 2017-02-14 11:50:20,815 Allocating for 4.194e+06 particles
Initializing coarse index at order 5: 100%|| 12/12 [00:00<00:00, 14.60it/s]
Initializing refined index at order 3: 100%|| 12/12 [00:01<00:00,  8.80it/s]
```

And produce a file named `my_index` in the same folder as `snapshot_033/snap_033.0.hdf5`. The second and all later times the script is run we only need to load the index from disk, so it produces the following output:

```
yt : [INFO     ] 2017-02-14 11:56:07,977 Allocating for 4.194e+06 particles
Loading particle index: 100%|| 12/12 [00:00<00:00, 636.33it/s]
```

Note that there 12 iterations for each loop. Each of these iterations correspond to a single IO chunk. If a file has fewer than 262144 particles, the entire file is used as an IO chunk. If a file has more than 262144 particles, the file is logically split into several subfiles, each containing up to 262144 particles. Currently the chunk size of 262144 particles is hard-coded for all SPH frontends.

Changes to the Selector API

The Morton bitmaps needed for individual data objects are constructed using the existing low-level Cython selection API. To determine whether a given Morton index is "contained" in the geometric primitive defined by the selector we make use of the `select_bbox` selection API call, since each index corresponds to a single cell in an octree. If the selector fully encloses the bounding box for the cell defined by a given Morton index, the existing `select_bbox` function is sufficient. However, given that the goal of the Morton bitmap index is to reduce the number of files we need to read from for a given selection operation, more care must be taken near the "edges" of a selector. For this reason, we have added a new function to the selector API, `select_bbox_edge`. This function is identical to `select_bbox` in the case when a bounding box is fully contained inside of the geometric primitive associated with a selector, simply returning 1 in these cases. However, if the bounding box is only partially contained in the geometric primitive, `select_bbox_edge` returns 2, indicating partial overlap. This is used in the bitmap index code to indicate that the coarse Morton index does not have sufficient resolution in this region, triggering the generation of refined Morton indices in this region. These smaller bounding boxes will have a higher probability of being either fully contained or fully excluded from a data object, decreasing the probability of a file collision. The `select_bbox_edge` function has been implemented for all selectors and if this YTEP is accepted will be a required part of the API for new selectors in the future.

In addition to the above change, a more minor change was necessary to the portion of the selector API used to count and select particles contained in a given selector. Currently, all particles are assumed to be pointlike, which will lead to incorrect selections for particles that actually have finite volumes like SPH particles. To account for this, the signature of the `count_points` and `select_points` functions were changed so that instead of accepting only single scalar radius for all particles, they can accept an array of possibly variable radii as well. If non-zero radii are passed in, particle selection operates via the `select_sphere` method instead of the `select_point` method that is currently used. Since some selectors did not yet have implementations of `select_sphere`, we have added new implementations where necessary.

Testing

Testing is provided for both the low level routines controlling access to the bitmap, as well for higher level routines that control bitmap generation. Low level tests are located in *yt.utilities.lib.tests.test_geometry_utils*. This includes tests of the routines for generating Morton indices from cartesian coordinates, extracting single bit coordinates, and locating neighboring morton indices at the coarse or refined index level both with and without periodic boundary conditions.

Higher level tests are located in *yt.geometry.tests.test_particle_octree* and include the framework to create test datasets with arbitrary domain decomposition schemes across a specified number of files. Tests are included for creating bitmap indices for datasets with no collisions and all collisions that check the number of coarse and refined cells against known answers. In addition we also provided tests for saving/loading bitmaps and identifying input files for rectangular selections on known domain decomposition schemes.

31.4.3 Removing the Global Octree Mesh

Currently, all I/O operations are mediated via the global octree index. Particles are read in from the output file as needed based on their position in the octree. With the arrival of the compressed bitmap index scheme described above, we no longer need to use the global octree to manage I/O chunking. Making the global octree redundant in this way raises the question about whether the octree is really needed at all.

Currently yt makes a distinction between particle fields and mesh fields. All SPH-smoothed fields (e.g. `('gas', 'density')`) are smoothed onto the global octree mesh. To make a concrete example, let's try loading an SPH zoom-in simulation of a galaxy and ask for the `('gas', 'density')` field:

```
import yt
ds = yt.load('GadgetDiskGalaxy/snapshot_200.hdf5')

ad = ds.all_data()
density = ad['gas', 'density']

print(density.shape)
print(ds.particle_type_counts)
```

Running this script on the latest development version of yt at time of writing (`abf5a8eff1b2`) produces the following output:

```
(5661944,)
{'PartType0': 4334546,
 'PartType1': 4786616,
 'PartType2': 2333848,
 'PartType3': 0,
 'PartType4': 450921,
 'PartType5': 1149}
```

On my laptop, this script also takes about 116 seconds to run, with 105 s spent performing the SPH smoothing operation onto the global octree. Note also how the number of leaf octs in the octree (5661944) does not match the number of SPH particles (`PartType0`). This discrepancy is a common source of initial confusion for users of SPH codes when they first try to use yt to analyze their data.

We can ask ourselves whether it makes sense to always smooth data onto the global octree. It makes intuitive sense for users of AMR codes for yt to return data defined on a volume-filling mesh, since the volume filling mesh is the "real" data. However, for SPH data, the global octree mesh is not representative of the "native" data. By making the return value of most yt operations for SPH fields be defined on the octree mesh, yt is not being "true" to the data and also makes it harder than it needs to be to access the particle data as such.

In this YTEP, **we propose changing the data object API for SPH data by ensuring that all SPH smoothed fields return data defined at the locations of SPH particles**. This means that rather than relying on smoothing data onto

the global octree, we will instead always return data defined at the particle locations. This means that running the script included above would produce the following output:

```
(4334546,)
{'PartType0': 4334546,
 'PartType1': 4786616,
 'PartType2': 2333848,
 'PartType3': 0,
 'PartType4': 450921,
 'PartType5': 1149}
```

And that the `('gas', 'density')` field would merely be an alias to the `('PartType0', 'Density')` field available on-disk. Since we no longer need to smooth data onto the in-memory global octree, this substantially reduces the memory needed to work with SPH data while simultaneously substantially improving performance. Just as an example, in the version of the yt that implements this YTEP, the script at the top of this section requires only 3.3 seconds to run.

The details of how this backward incompatible change to the yt user experience for SPH data will be implemented is detailed below. This includes all design decisions that have been made in the prototype version of yt that implements this YTEP. In addition, there are still several design decisions about how to implement this YTEP that have not yet been decided on. For more details about these issues, see the "Open Questions" section at the bottom of this document.

Identifying the SPH Particle

All of the proposals in this YTEP require that there be special handling for fields that correspond to the SPH particle type. Currently yt does not have a way of identifying whether a given particle type in a particle dataset is an SPH particle. To ameliorate this, we propose adding a new private attribute of `SPHDataset` instances, `_sph_ptype`. This attribute should resolve to the string name of the SPH particle type for the given output type. For example, for Gadget HDF5 data, the `_sph_ptype` is `'PartType0'`. Having this attribute available makes it much easier to write code that does special handling for SPH data.

SPH Fields

Here we discuss changes to the yt field system for SPH particle data that will enable removing the global octree mesh.

Local Fields

Currently yt assumes that fields with a `'gas'` field type are defined on a volume filling mesh. This YTEP proposes relaxing that assumption for SPH data so that `'gas'` fields correspond to *particle* fields. Since we would like to reuse the existing field definitions in yt as much as possible, we need to explore how to adjust the field system to allow reuse of existing fields when the field data might represent local particle data, SPH smoothed quantities, or mesh fields, depending on the type of data being loaded.

As a reminder, `sampling_type` is a newly introduced keyword argument that can be passed to the initializer for yt `DerivedField` objects that will be released publicly as part of yt 3.4. It replaces the `particle_type` keyword argument, allowing more flexibility to define new types of fields that are sampled in novel ways without needing to expose additional keyword arguments like `particle_type`. Currently, the default value of `sampling_type` is `'cell'`, preserving the old default behavior (e.g. `particle_type=False`).

We propose changing the default value of the `sampling_type` used for yt derived fields from `'cell'` to a new value: `'local'`. Derived fields with `sampling_type='local'` are fully local functions of other derived fields (which themselves do not have to be fully local). It turns out that nearly all of the fields that are currently defined inside yt with `sampling_type='cell'` are actually fully local and the field functions they encode can be readily

reused with particle data. In the version of yt that implements this YTEP, all fully local derived fields defined inside yt have had their field definitions altered such that `sampling_type='local'`.

With this accomplished, making all fully local derived fields work simply requires setting up SPH particle fields with aliases to yt "universal" field names. To make that concrete, this means that a Gadget HDF5 output needs an alias from `('PartType0', 'Density')` to `('gas', 'density')`. With this alias defined, all fully local derived fields that depend only on `('gas', 'density')` will automatically work. In addition, any *particle* derived fields defined for the `PartType0` with field names that begin with `'particle_'` will be aliased to `'gas'` fields without the `'particle_` prefix. For example, the `('PartType0', 'particle_angular_momentum_x')` field is aliased to `('gas', 'angular_momentum_x')`. This means that any `'gas'` derived fields that depend on `('gas', 'angular_momentum_x')` being defined will function as expected. In other words, we use the existing system of particle fields to bootstrap the needed "input" fields for the bulk of the `'gas'` derived fields. The aliasing described here is implemented in the `setup_smoothed_fields` member function of the `FieldInfoContainer` class.

One side effect of this approach is that there are some "odd" `'gas'` derived fields (particularly if one is coming from an AMR code). For example, `('gas', 'position')` is defined as an alias to `('PartType0', 'particle_position')`. It may not be a good idea in the end to alias **all** particle fields for the SPH particle type to `'gas'` fields, and it may be necessary to add a blacklist of fields that should *not* be aliased, or that should be aliased with explicit particle field names (e.g. maybe it would be most helpful to define `('gas', 'particle_position')`).

Non-local Fields

Unfortunately, not all fields are fully local. We would optimally like to support fields that require some sort of difference operation, in particular physically meaningful fields like the gas vorticity or divergence. Currently these fields are not supported for particle data (since ghost zones have not yet been implemented for octrees), so if this effort makes it easier to add support for these fields, that will be a substantial improvement.

It turns out that within the SPH formalism there is a straightforward way to compute fields that depend on spatial derivatives. These formulae are used internally in SPH codes to estimate various terms in the equations of fluid dynamics. Thankfully, we can make use of these formulae for visualization and analysis purposes. There is a very nice paper by Dan Price [PricePaper] that works through this formalism, from which we can derive several formulae for partial derivatives and vector derivatives. For some quantity A that is a function of position, the partial derivative of A with respect to x at the position of particle a can be evaluated via:

$$\frac{\partial A_a}{\partial x} = \sum_b \frac{m_b}{\rho_b} \frac{\phi_b}{\phi_a} \left(A_b - A_a\right) \frac{\partial_a W_{ab}}{\partial x}$$

Here m_b and ρ_b are the mass of and gas density associated with the b'th particle, ϕ is an arbitrary function of position (common choices are 1 and ρ), and W_a is the SPH smoothing kernel at the position of particle a. The derivative inside the sum in the above expression is evaluated at the position of particle a.

Similarly for the gradient, divergence, and curl:

$$\nabla_a A = \sum_b \frac{m_b}{\rho_b} \frac{\phi_b}{\phi_a} \left(A_b - A_a\right) \nabla_a W_{ab}$$

$$\left\langle \nabla \cdot \mathbf{A} \right\rangle_a = \sum_b \frac{m_b}{\rho_b} \frac{\phi_b}{\phi_a} \left(\mathbf{A}_b - \mathbf{A}_a\right) \cdot \nabla_a W_{ab}$$

$$\left\langle \nabla \times \mathbf{A} \right\rangle_a = -\sum_b \frac{m_b}{\rho_b} \frac{\phi_b}{\phi_a} \left(\mathbf{A}_b - \mathbf{A}_a\right) \times \nabla_a W_{ab}$$

These symmetrized formulae (i.e. they all include a term that looks like $A_b - A_a$) have the advantage that the derivative of a constant field is zero by construction.

To actually use these formula, we will need to calculate on a particle-by-particle basis the list of nearest neighbors for each particle and then evaluate these formulae at the locations of each particle. This has not yet been implemented in the version of yt that implements this YTEP, but we expect it to be straightforward using the existing functionality in yt to generate nearest neighbor lists.

Non-local fields that do not depend on an explicit derivative operation will (e.g. `('gas', 'averaged_density')`) will not be implemented for SPH data.

Data Selection for SPH Fields

Currently data selection for particle fields models all particles, including SPH particles, as infinitesimal points. This means that 2D data objects do not select particles without exact floating point intersection between the data object and the particle.

This YTEP proposes modifying the selection semantics for SPH particles. Instead of modeling SPH particles as infinitesimal points, we will select SPH particles if the smoothing volume intersects with the data container. This means that particles with positions outside of 3D data containers will be selected, since if the smoothing volume overlaps these particles still contribute to estimates of fluid quantities inside of the data object. See *Testing* for more discussion of the testing strategy used to validate the yt data object and data selection system for SPH particles. To allow for use cases where it is convenient to treat SPH particles as infinitesimal, we will add the ability to dynamically turn on and off this behavior with a configuration option.

We have implemented and added unit tests for all of the following data objects:

- Point
- Slice
- Off-axis Slice
- Region
- Disk
- Ray

In addition, we have implemented the following data object features that depend on hooks in the C selector API:

- Chained selection (e.g. `reg = ds.region(..., data_source=sphere)`)
- Boolean negation
- Boolean addition
- Boolean AND
- Boolean XOR
- `ds.intersection`
- `ds.union`

Visualization of Slices and Projection

Currently slices and projections of SPH data are generated by slicing data that has been SPH smoothed onto the global octree. If there is no more global octree, an alternative strategy for generating pixelized representations of SPH data needs to be implemented. This YTEP proposes replacing the pixelizer operations for slices, off-axis slices, and axis-aligned projections to make use of an SPH-centric pixelization operation. For inspiration, we look to SPLASH, an open-source SPH visualization tool written in Fortran. The algorithms used in SPLASH are detailed in the SPLASH method paper [SPLASHPaper]. We note that we have not consulted the SPLASH source code in support of this implementation.

The key to the pixelization algorithm used in SPLASH is to compute the SPH smoothing operation via the "scatter" approach. Rather than looping over pixels in the image, determining which particles contribute to the SPH smoothing operation at the location of that pixel, and then compute a field value using the SPH smoothing formula, we instead loop over particles, finding the set of pixels whose smoothing volumes overlap with the pixel location and deposit a contribution for that particle to all of the pixels the smoothing volume overlaps. As we loop over all of the particles that contribute to the image, we fill in the image by summing the contributions of each particle. This approach is attractive because it does not require any sort of nearest-neighbor operation and is also trivially parallelizable using e.g. OpenMP threads.

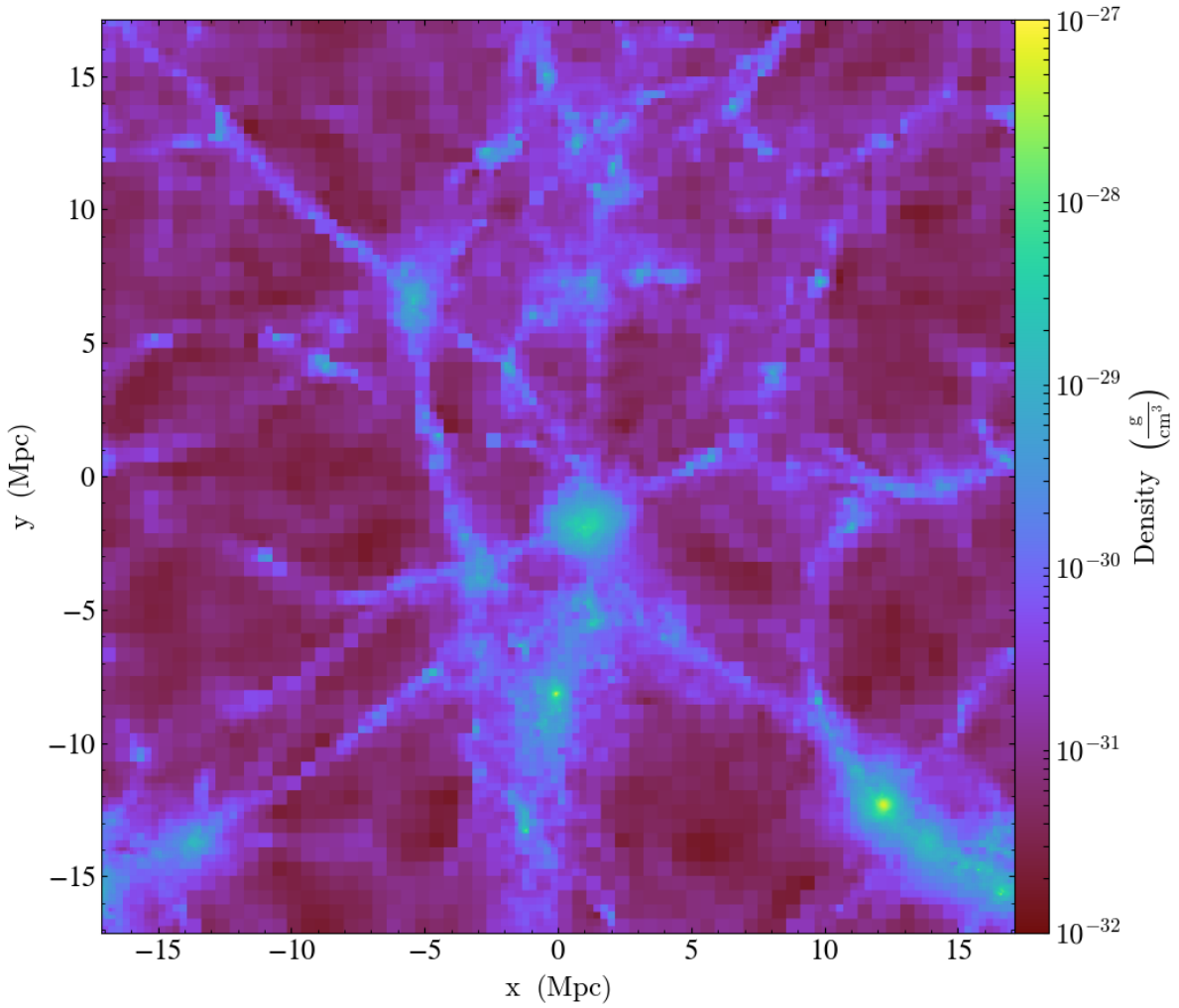
For slices we estimate the contributions of a particle to a single pixel using the standard SPH smoothing formula. For Projections we make use of a projected version of the smoothing formula, taking advantage of the spherical symmetry of the problem. The smoothing operation is implemented in two Cython functions: `pixelize_sph_kernel_slice` and `pixelize_sph_kernel_projection` which are defined in `yt.utilities.lib.pixelization_routines`.

To make the above discussion a bit more concrete, consider the following script:

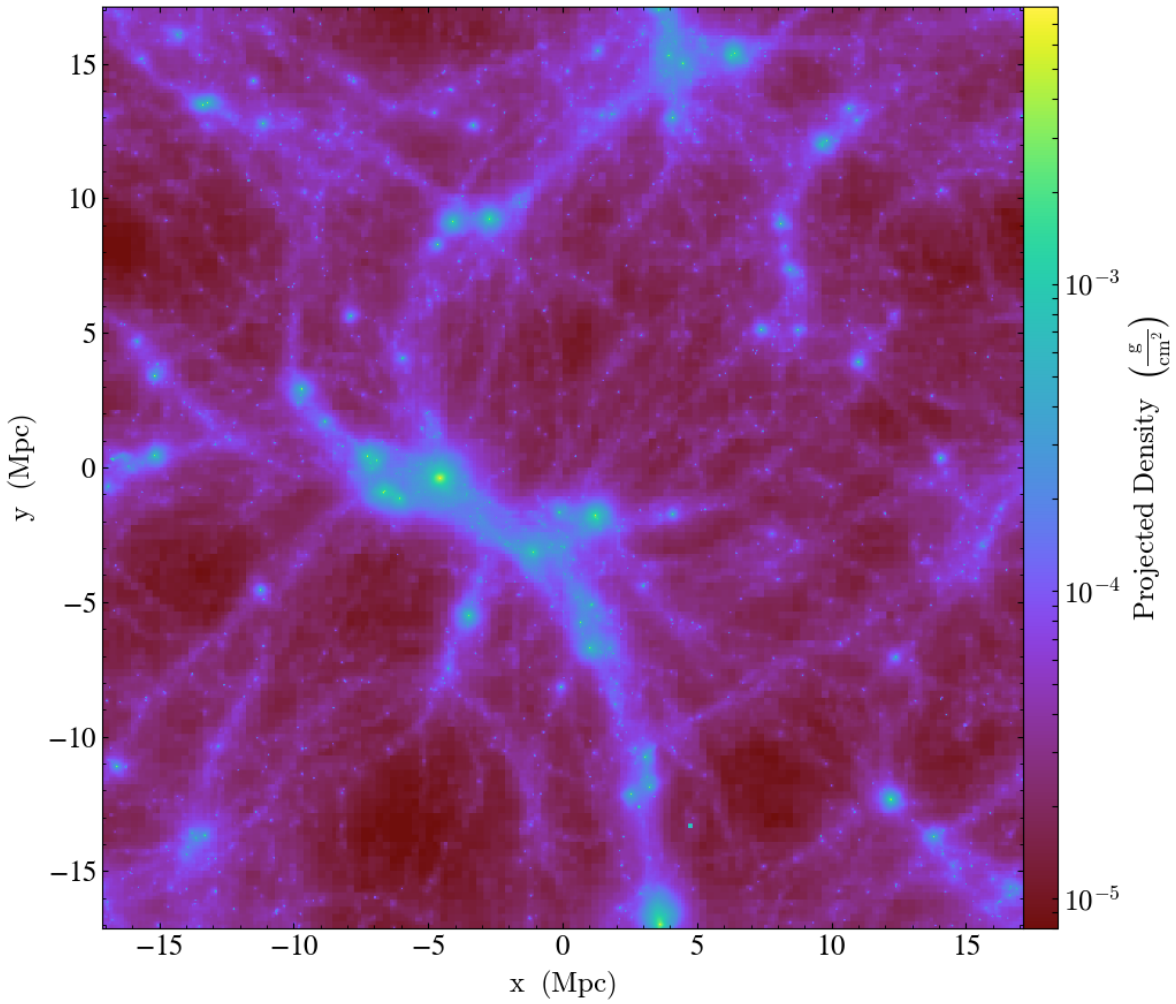
```
import yt

ds = yt.load('snapshot_033/snap_033.0.hdf5')

plot = yt.SlicePlot(ds, 2, ('gas', 'density'))

plot.set_zlim(('gas', 'density'), 1e-32, 1e-27)

plot.save()

plot = yt.ProjectionPlot(ds, 2, ('gas', 'density'))

plot.set_zlim(('gas', 'density'), 8e-6, 8e-3)

plot.save()
```

Running the latest development version of yt at time of writing (`25651334863b`) requires 43 seconds to run and produces the following images:

Slice:

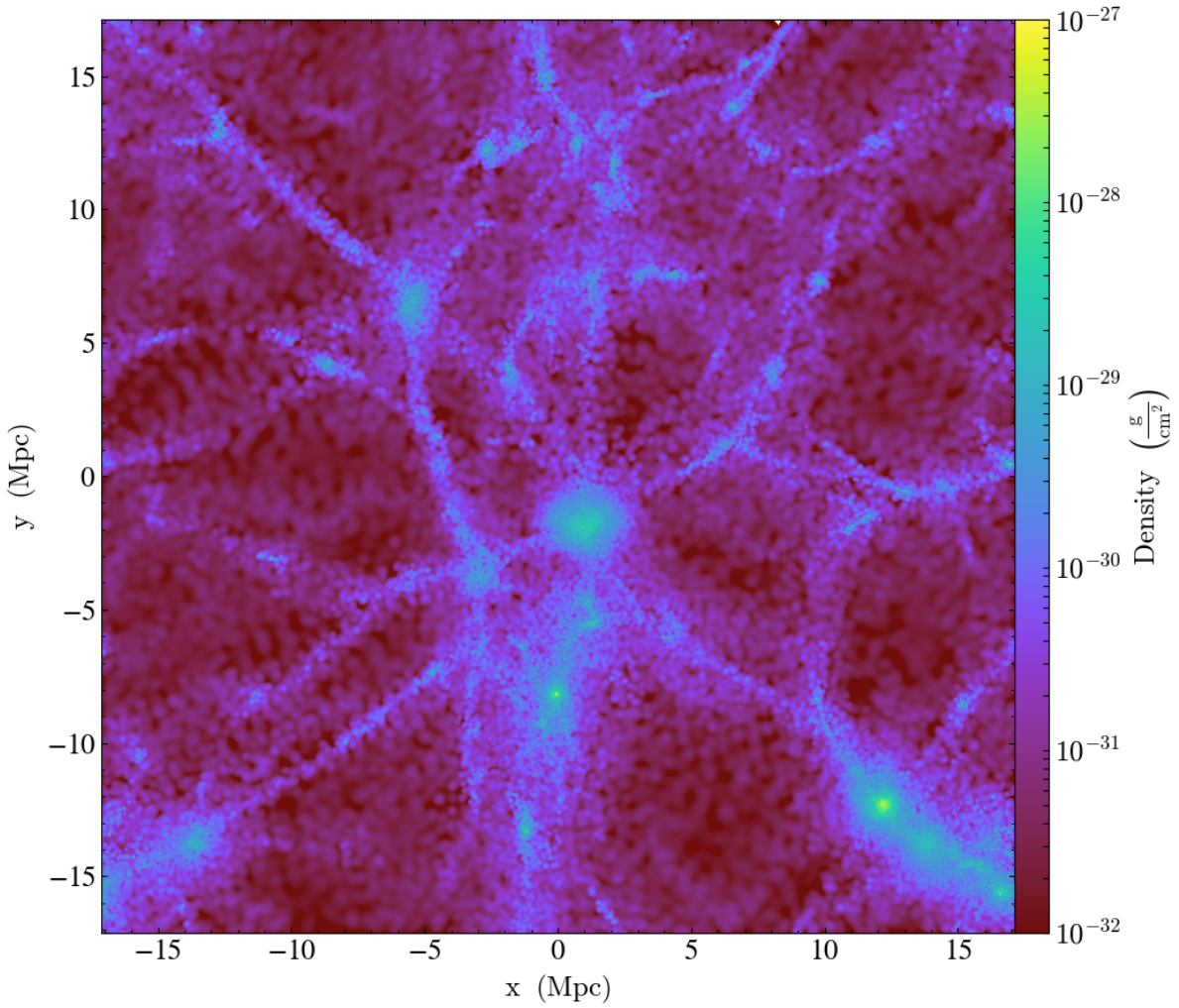

15
10
5
0
−5
−10
−15
y (Mpc)
−15
−10
−5
0
5
10
15
x (Mpc)
10^{-27}
10^{-28}
10^{-29}
10^{-30}
10^{-31}
10^{-32}
Density $\left(\frac{g}{cm^3}\right)$

Projection:

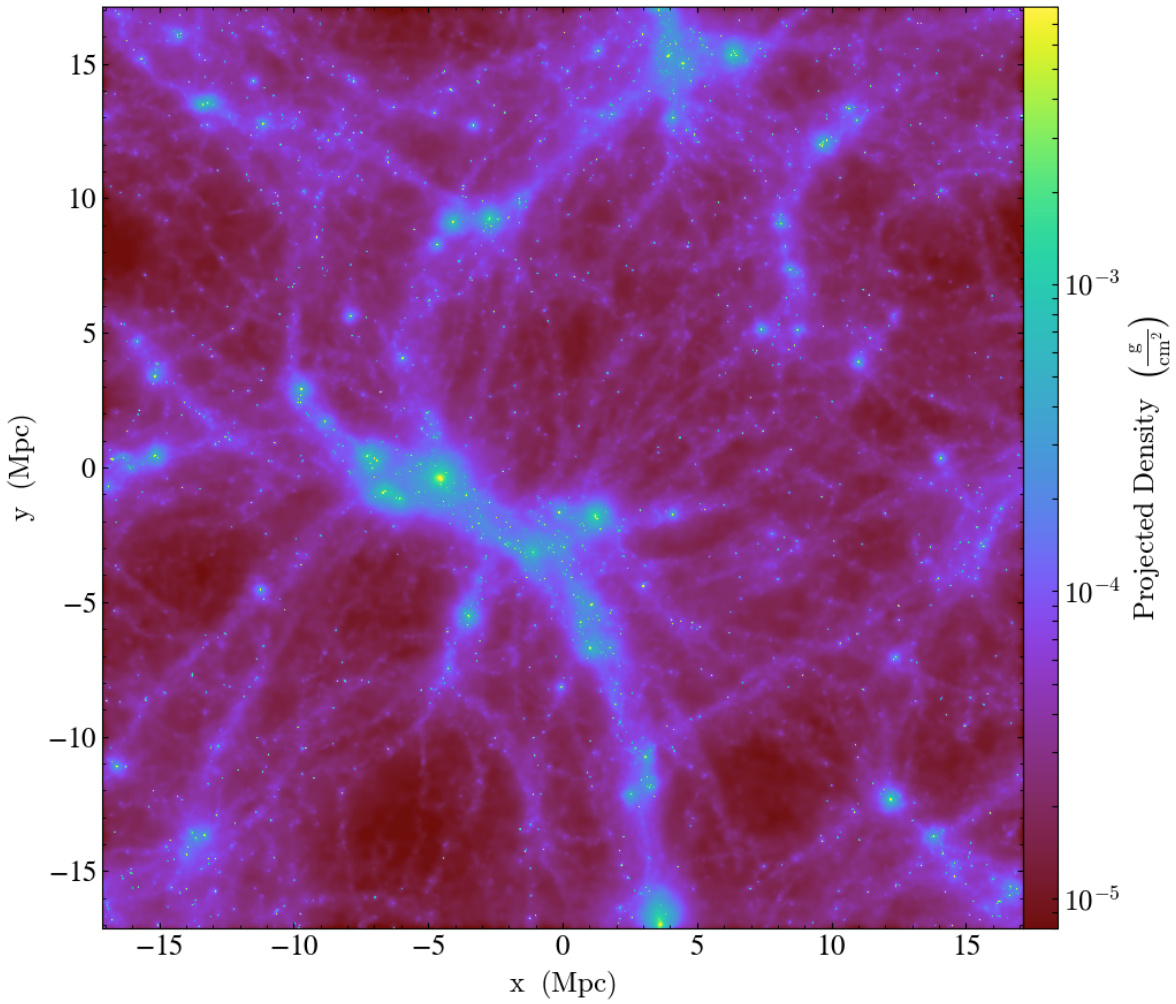


Running the same script on the version of yt that implements this YTEP produces requires 20 seconds and produces the following images:

Slice:


15
10
5
0
−5
−10
−15
y (Mpc)
−15
−10
−5
0
5
10
15
x (Mpc)
10^{-27}
10^{-28}
10^{-29}
10^{-30}
10^{-31}
10^{-32}
Density $\left(\frac{g}{cm^2}\right)$

Projection:



Note also that the performance improvement here becomes more stark for larger datasets as well as for zoom-in simulations which have deeper octrees.

The images produced using the octree are quite "blocky", since the resolution of the image in any given location is limited by the octree. This could be ameliorated somewhat using `over_refine_factor` but that requires steeper memory and runtime cost requirements to smooth onto the octree. In general the images produced by the new pixelizers are truer to the actual structure of the data. Rather than generating an image from a sampled representation of the real data, it is our opinion that it makes more sense to instead sample directly from the particle data.

Deposition operations

Regular Grids

While we do want to make it easier to access particle-centric data, we need to make sure it's still possible to locally deposit and SPH smooth data onto grids. Not only is that a useful operation for users of SPH codes, but it's also functionality that yt currently provides, so we need to ensure that currently supported operations on `covering_grid` and `arbitrary_grid` data objects continue to work and produce sensible results. We will add tests to verify that this is the case.

Octrees

We should not abandon the ability to smooth SPH data and deposit particle data onto a volume-filling octree. Simply because users are currently using these data for their own analyses, we need to provide a migration path so that users can reproduce prior work made with yt using the global octree.

We propose adding a new data object to yt that represents an octree with a given bounding box (which need not overlap with the domain bounding box) and maximum refinement level. One can think of this as something of an adaptive `arbitrary_grid` data object. Initially we will only allow refinement in terms of particle quantities (e.g. particle mass or particle count per octree leaf node), but it should be possible to add support for data defined on octree or patch AMR meshes eventually.

We still need to decide on an appropriate API for this. Ideally we would be able to reuse some of the existing code for the global octree.

Testing

The testing strategy for this work follows two basic approaches so far. First, we make sure that all derived fields that are associated with a number of real-world SPH datasets from http://yt-project.org/data can be calculated without generating any errors. This ensures both that the derived field system is functioning but also that the I/O routines in the various SPH frontends are functioning correctly. These tests are present in `yt.fields.tests.test_sph_fields`.

In addition, we have added support in the stream frontend for loading SPH data. This allows us to create fake in-memory SPH datasets that we can construct in a way that make them easier to reason about for testing than a real-world SPH dataset. The primary route for generating these dataset is a new function in the `yt.testing` namespace, `fake_sph_orientation_ds`. This function has the following very straightforward definition:

```
def fake_sph_orientation_ds():
    """Returns an in-memory SPH dataset useful for testing

    This dataset should have one particle at the origin, one more particle
    along the x axis, two along y, and three along z. All particles will
    have non-overlapping smoothing regions with a radius of 0.25, masses of 1,
    and densities of 1, and zero velocity.
    """
    from yt import load_particles

    npart = 7

    # one particle at the origin, one particle along x-axis, two along y,
    # three along z
    data = {
        'particle_position_x': (
            np.array([0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]), 'cm'),
        'particle_position_y': (
            np.array([0.0, 0.0, 1.0, 2.0, 0.0, 0.0, 0.0]), 'cm'),
        'particle_position_z': (
            np.array([0.0, 0.0, 0.0, 0.0, 1.0, 2.0, 3.0]), 'cm'),
        'particle_mass': (np.ones(npart), 'g'),
        'particle_velocity_x': (np.zeros(npart), 'cm/s'),
        'particle_velocity_y': (np.zeros(npart), 'cm/s'),
        'particle_velocity_z': (np.zeros(npart), 'cm/s'),
        'smoothing_length': (0.25*np.ones(npart), 'cm'),
        'density': (np.ones(npart), 'g/cm**3'),
    }
```

(continues on next page)

(continued from previous page)

```
    bbox = np.array([[-4, 4], [-4, 4], [-4, 4]])

    return load_particles(data=data, length_unit=1.0, bbox=bbox)
```

This example also demonstrates how `load_particles` can be used by users in this work to load SPH data written in data formats that aren't yet supported by a frontend. This testing dataset has one particle at the origin, another particle along the x axis, two more along the y axis, and three along z. All particles have the same smoothing length, such that the smoothing volumes of any of the particles in the dataset do not overlap. This means that we can construct various data objects and reason about which particles we should be selecting given the geometry of the particles in the dataset and the boundaries of the data object. In addition, we take care to make sure that the boundaries of the data objects do not necessarily directly overlap with the position of a particle near the boundary. This ensures that particles are selected when their smoothing volume overlaps with a data object, not necessarily based on the particle positions. These tests are present in `yt.data_objects.tests.test_sph_data_objects`. Currently all of the data objects supported by yt are explicitly tested here. As an example, here is the test that verifies the slice data object is working correctly:

```
# The number of particles along each slice axis at that coordinate
SLICE_ANSWERS = {
    ('x', 0): 6,
    ('x', 0.5): 0,
    ('x', 1): 1,
    ('y', 0): 5,
    ('y', 1): 1,
    ('y', 2): 1,
    ('z', 0): 4,
    ('z', 1): 1,
    ('z', 2): 1,
    ('z', 3): 1,
}

def test_slice():
    ds = fake_sph_orientation_ds()
    for (ax, coord), answer in SLICE_ANSWERS.items():
        # test that we can still select particles even if we offset the slice
        # within each particle's smoothing volume
        for i in range(-1, 2):
            sl = ds.slice(ax, coord + i*0.1)
            assert_equal(sl['gas', 'density'].shape[0], answer)
```

31.5 Open Questions

There are a number of design decisions that still need to be made if this YTEP is going to be fully implemented and accepted. Comments and suggestions on these points are very welcome.

31.5.1 The Projection Data Object

Currently the projection data object is completely broken for particle data for all frontends:

```
In [1]: import yt

In [2]: ds = yt.load('IsolatedGalaxy/galaxy0030/galaxy0030')
yt : [INFO     ] 2017-03-01 09:54:22,491 Parameters: current_time              = 0.
↪0060000200028298
```

(continues on next page)

(continued from previous page)

```
yt : [INFO     ] 2017-03-01 09:54:22,491 Parameters: domain_dimensions         = [32␣
↪32 32]
yt : [INFO     ] 2017-03-01 09:54:22,492 Parameters: domain_left_edge          = [ 0.␣
↪ 0.  0.]
yt : [INFO     ] 2017-03-01 09:54:22,492 Parameters: domain_right_edge         = [ 1.␣
↪ 1.  1.]
yt : [INFO     ] 2017-03-01 09:54:22,492 Parameters: cosmological_simulation   = 0.0

In [3]: proj = ds.proj(('gas', 'density'), 0)
Parsing Hierarchy : 100%|| 173/173 [00:00<00:00, 3535.74it/s]
yt : [INFO     ] 2017-03-01 09:54:27,650 Gathering a field list (this may take a␣
↪moment.)
yt : [INFO     ] 2017-03-01 09:54:29,653 Projection completed

In [4]: proj['all', 'particle_mass']
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-1a26d598985b> in <module>()
----> 1 proj['all', 'particle_mass']

/Users/goldbaum/Documents/yt-hg/yt/data_objects/data_containers.py in __getitem__
↪(self, key)
    281                 return self.field_data[f]
    282             else:
--> 283                 self.get_data(f)
    284         # fi.units is the unit expression string. We depend on the registry
    285         # hanging off the dataset to define this unit object.

/Users/goldbaum/Documents/yt-hg/yt/data_objects/construction_data_containers.py in␣
↪get_data(self, fields)
    339                     self._initialize_projected_units(fields, chunk)
    340                     _units_initialized = True
--> 341                 self._handle_chunk(chunk, fields, tree)
    342         # Note that this will briefly double RAM usage
    343         if self.method == "mip":

/Users/goldbaum/Documents/yt-hg/yt/data_objects/construction_data_containers.py in _
↪handle_chunk(self, chunk, fields, tree)
    440         v = np.empty((chunk.ires.size, len(fields)), dtype="float64")
    441         for i, field in enumerate(fields):
--> 442             d = chunk[field] * dl
    443             v[:,i] = d
    444         if self.weight_field is not None:

/Users/goldbaum/Documents/yt-hg/yt/units/yt_array.py in __mul__(self, right_object)
    955         """
    956         ro = sanitize_units_mul(self, right_object)
--> 957         return super(YTArray, self).__mul__(ro)
    958
    959     def __rmul__(self, left_object):

ValueError: operands could not be broadcast together with shapes (3976,) (37432,)
```

By making SPH data return most data as particle fields we are making this problem much more visible. We should decide what a sensible return value for the projection operation on a particle field should be. Note that in practice we do not need to solve this issue to create a `ProjectionPlot` since we can short-circuit the data selection operation when we create the pixelized projection.

Some ideas:

- Write a new `ParticleQuadTree` class that adaptively deposits particle data onto a quadtree mesh.
- Since most people really want a pixelized representation of the particle data (e.g. via a `FixedResolutionBuffer` we could simply make it so the projection data object returns a regular resolution image.

31.5.2 Cut Regions

We have not yet implemented the Cut Region data object since it's not clear how it should work for particle data. Similar to the projection data object, the cut region data object does not currently work when it is defined in terms of a threshold on a particle field. It may be possible to define an internal particle filter to implement cuts on Lagrangian data.

31.5.3 Volume Rendering

Currently we don't support volume rendering particle data. In principle writing at least a basic volume renderer specifically for particle data is a straightforward project. Making it scale well to arbitrarily large datasets would be a bigger undertaking, but we think we should attempt to write a volume rendering engine that accepts particle data. Optimally this will plug in to the existing volume rendering infrastructure at the same level as the AMRKDTree. Attempting this will also make it easier to add support for volume rendering octree AMR data with an octree volume rendering engine.

The API and design for this component have not yet been settled.

31.6 Community engagement

We are early enough in the process of implementing this YTEP that the major design points have not yet calcified. To encourage wide adoption of these changes with a minimum amount of breakage for existing users, we will to reach out to existing users of yt who regularly work with SPH data to ensure that their existing code continue to work as much as possible. If there are widespread breakages, this will inform where we should focus on building backward compatibility shims and helpers.

Before this work can be merged into the main development branch we will need to update the documentation for particle data. This should include coverage of the following topics:

- Loading in-memory SPH data using `load_particles`
- A high-level description of the Morton bitmap index system and how to tune it for performance by adjusting the maximum coarse and refined Morton index level.
- A high-level description of the data selection semantics for particle and SPH data.
- A transition guide explaining all of the changes and how to port scripts, particularly those making direct use of the global octree via a deposition operation.

Finally, we will attempt to publicize this document as much as possible to attract feedback from current and prospective users at an early stage.

31.6.1 yt 4.0?

Since these are substantial backward incompatible changes, we think the next version of yt released after this work is merged should be yt 4.0. This opens the possibility of adding other backward incompatible changes as well as

removing deprecated features. We should be sure to signal to our users that there will only be major changes for those who work with SPH data - support for AMR and unstructured mesh data should remain the same.

CHAPTER 32

YTEP-0033: Dropping Python2 Support

32.1 Abstract

Created: November 28, 2017 Author: Nathan Goldbaum Revision: Matt Turk

We will be dropping support for Python 2.7 in all "major" releases after 3.5, which will include both 3.6 and 4.0.

32.2 Status

Completed

32.3 Project Management Links

This has come up in the past on the mailing list:

http://lists.spacepope.org/pipermail/yt-dev-spacepope.org/2017-April/006851.html

This document formalizes much of the discussion in that mailing list thread.

32.4 Detailed Description

32.4.1 Background

Python 2.7, the last major release in the Python 2 series, is currently in maintenance mode, only receiving critical bugfixes. Even this minimal level of support will end on April 12, 2020, when Python 2.7 will go end of life. At that point, Python 2.7 will no longer receive even security fixes. It is thus incumbent on us to not encourage our users to use an insecure, unsupported version of Python that may not even necessarily build on future versions of operating systems by maintaining support for Python 2.7 indefinitely.

Beyond that, for purely selfish reasons, we as a community have taken on costs to support both Python 2 and Python 3 in the same codebase. All contributions must happen in the dialect of Python that simultaneously functions under both major versions. We make heavy use of `six`, requiring contributors to learn about `six` to make even trivial contributions that happen to touch on Python 2/3 incompatibilities. We must run tests on both versions to avoid version-specific bugs and regressions. We are unable to use new features that have been added in Python 3 since these features are unavailable in Python 2.

For this reason, many projects in the scientific python ecosystem have either already ended support or proposed ending support for Python 2.7 in the near future. These efforts are summarized in the Python 3 statement, including a timeline that depicts the end of Python 2.7 support for projects that have signed the statement.

32.4.2 Proposed Solution

This YTEP proposes that yt 3.5 be the last major release of yt that will support Python 2.7. Subsequent releases, including 3.6 and 4.0 (but not including 3.5.1, 3.5.2, etc) will not support Python 2.7. In the past, we had suggested that yt 3.6 would not exist, or would not drop python 2.7 support, but the delay in yt 4.0 release has changed this timeline.

The development bandwidth for yt is somewhat restricted. In the past, we have had to ensure installation and utilization on many reasonably slow-moving environments (in particular HPC) but in the last five years or so the change in environments on HPC resources has largely eliminated this as a concern; we do not need to ensure extremely long-term compatibility with python 2.7. Some projects are treating their last release that supports Python 2.7 as a "long term support" (LTS) release. Given our development resources, we generally do not have the bandwidth available to simultaneously support both yt 4.0 and and python2-enabled yt 3.5 at the same level.

We are thus proposing that we make "best effort" compatibility with python2 in the 3.6 series (i.e., "soft" backcompat) and we will continue to accept patches for 3.5, although we anticipate these will taper with time, and we will cease to accept them one calendar year after support for python2.7 has ended. yt 4.0 will not have any backcompat built in.

To summarize:

- yt 3.5: This will be the last release series that explicitly supports python 2. No further development is anticipated once yt 3.6 has been released.
- yt 3.6: No hard python2 compatibility requirements, and "best efforts" may not be retained over the development lifetime. Testing of python2 will be dropped prior to release.
- yt 4.0: No back-compat with python 2.

These timelines are roughly comparable with projects that we directly depend on. In particular:

- SymPy: Dropping support in 2019.
- Numpy: Dropping support January 1 2019
- Matplotlib: Dropping support in 2018

In principle we don't need to be constrained by the timelines of projects we depend on, since we need only use the last version that supports Python 2.7, but that would add yet another maintenance burden, since we would not be able to use the latest and greatest version of a downstream project for development.

Since we will be backporting bugfixes to the yt 3.6 release branch while yt 4.0 is under development, we will keep the uses of `six` and other python 2/3 compatibility code in an effort to make backporting easier. Once yt 4.0 is released we will be able to begin removing compatibility shims and `__future__` imports. We will be able to immediately drop the tests for Python 2.7 support on th development branch.

32.4.3 Community Outreach

I am planning to make both yt-dev and yt-users aware of this proposal, with the hope of soliciting feedback from interested users who do not normally participate in development discussions. Additionally, if this proposal is accepted, we will announce the future timeline for releases, along with the planned timeline for dropping Python 2.7 support in all release announcements. This should give more than a year of warning for community members to either say their peace about Python 2.7 support, or to make preparations to migrate user scripts to use Python 3.

32.5 Backwards Compatibility

Users who must remain on Python 2.7 for whatever reason will no longer be able to run the latest version of yt. Existing versions will continue to function, however. Users who currently run yt under python 3 (this is the default option in the install script since April) will see no change.

32.6 Alternatives

- Maintain support for Python 2.7 indefinitely.
- Maintain support for Python 2.7 for the yt 4.x series, dropping support in yt 5.0 or in a subsequent verison. This would likely mean maintaining support beyond 2020.

CHAPTER 33

YTEP-0034: yt FITS Image Standard

33.1 Abstract

Created: September 9, 2018 Author: John ZuHone

This YTEP will define the standard for `FITSImageData` objects written from slices, projections, and covering grids, for better support for reading these objects back into yt as datasets using the FITS frontend, using the dataset class `YTFITSDataset`.

33.2 Status

In Progress.

33.3 Project Management Links

The relevant code has been written and is in a PR which is under review:

https://github.com/yt-project/yt/pull/2010

33.4 Detailed Description

From its beginning, yt has been capable of producing projected images of simulations as representations of the quantities which would be observed on the sky plane. This enables comparisons of simulation predictions to real data. Nearly all observational data in astronomy is in the Flexible Image Transport System (FITS) format. Therefore, yt also has a method to write slices, projections, and regularly gridded data derived from datasets to FITS files using the `FITSImageData` class. Documentation on how to use this class and its subclasses may be found here.

FITS files consist of a list of "header data units" (hereafter HDUs), each of which contain data in image (an array of n dimensions with $n \geq 2$) or table form, associated with a header which typically contains information about the

coordinate system and other metadata. The header provides an opportunity to standardize FITS files written by yt so that the data is as self-describing as possible with respect to coordinates, units, and fields.

yt has also long had the capability to read FITS image files as datasets using the FITS frontend and the `FITSDataset` class. In general, each FITS HDU is classified as a yt field, and the metadata in its header is used to define the properties of the dataset. However, there is no universal standard for FITS files, and therefore in most cases a number of properties of these datasets may be undefined (e.g., units, coordinates, etc.).

At the very least, the FITS files produced by yt should be standardized. This requires ensuring that both the `FITSImageData` class and a new subclass of `FITSDataset`, `YTFITSDataset`, adhere to this standard. This YTEP serves to define the "yt FITS standard" for FITS images produced from 2/3D datasets using the `FITSImageData` class in yt and its subclasses.

33.5 Overall File Structure

yt FITS images shall be a single FITS file with one or more image HDUs, each one containing a 2 or 3-dimensional array with will correspond to a "field" in standard yt parlance. The dimensions of each array shall be consistent with the others for the entire file. The first or "primary" HDU will also contain an image array.

33.6 Header Information

Each FITS header associated with a a field in the FITS file shall be entirely self-describing with respect to the properties of the field, the current time of the dataset, the coordinate system of the dataset, and the unit system of the entire file. The file shall be distinguished as a yt FITS file by setting the `WCSNAME` property equal to `yt`.

33.6.1 Field Properties

The name of each field shall be stored in both the `EXTNAME` and `BTYPE` properties of the header. The units of the field shall be stored in the `BUNIT` property of the header.

33.6.2 Unit Information

Each header will be entirely self-describing as to the unit system of the dataset, including the dimensions of length, time, mass, velocity, temperature, and magnetic field units. In most cases, these units will be derived from the underlying dataset used by `FITSImageData` to produce the file, but it will be possible for the user to specify their own code unit definitions in the instantiation of the `FITSImageData` object. For images created by subclasses of `FITSImageData` such as `FITSSlice`, `FITSProjection`, etc., the `length_unit` of the file will be given by the units specified in the `width` keyword argument or be chosen automatically based on the size of the image.

33.6.3 Coordinate System

Each header shall have the coordinate system of the dataset stored in the WCS keywords. These will set up a linear coordinate system with an origin in pixel space at the center of the image. The relevant keyword arguments are:

- `CTYPE[123]`: The coordinate system type, all of which shall be `"LINEAR"` for all axes.
- `CUNIT[123]`: The units of the coordinate axes, in dimensions of length and in the specified `length_unit` of the FITS image. The units should be the same for all axes.
- `CRPIX[123]`: The reference pixel coordinate of the image, which shall always be the center of the image: `0.5*(n[xyz]+1)`, where `n[xyz]` is the number of pixels in each dimension.

- `CDELT[123]`: The width of each pixel along each axis in the units specified by `CUNIT[123]`.
- `CRVAL[123]`: The reference physical coordinate of the image, which corresponds to the same location as `CRPIX[123]`.

33.6.4 Other Metadata

Each header shall have the current time of the dataset stored in the header keyword `"TIME"`, where the units shall be the code time units of the dataset.

Future iterations of this standard may allow for other optional metadata such as the redshift, etc., which can be checked upon instantiation of the `YTFITSDataset` instance.

33.6.5 Example Header

The following is an example header for a density field created from a slice of a FLASH dataset, as printed out by the AstroPy command-line tool `fitsheader`:

```
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                  -64 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                  512
NAXIS2  =                  512
EXTEND  =                    T
EXTNAME = 'DENSITY '           / extension name
BTYPE   = 'density '
BUNIT   = 'g/cm**3 '
LUNIT   =                  1.0 / [kpc]
TUNIT   =                  1.0 / [s]
MUNIT   =                  1.0 / [g]
VUNIT   =                  1.0 / [cm/s]
BFUNIT  =    3.544907701811032 / [gauss]
TIME    = 1.18350909938232E+17
WCSAXES =                    2
CRPIX1  =                256.5
CRPIX2  =                256.5
CDELT1  =            0.9765625
CDELT2  =            0.9765625
CUNIT1  = 'kpc     '
CUNIT2  = 'kpc     '
CTYPE1  = 'LINEAR  '
CTYPE2  = 'LINEAR  '
CRVAL1  =                  0.0
CRVAL2  =                  0.0
LATPOLE =                 90.0
WCSNAME = 'yt      '
```

Many of the items in the header are automatically filled, but the rest are defined by yt.

33.7 Backwards Compatibility

FITS files generated using `FITSImageData` prior to these changes will still be readable, and may be recognizable as `YTFITSDataset` objects if they have the `"WCSNAME"` keyword set to `"yt"` in the FITS header. If not, they will still be readable as generic `FITSDataset` objects as before. Since previous FITS files made with `FITSImageData`

did not include unit information in their headers, units for these files will back to default cgs values if recognized as `YTFITSDataset` instances by yt.

33.8 Alternatives

Leaving things the way they are, which means that we will have support for writing `FITSImageData` objects to FITS files which can be read in and *mostly* understood by yt with the currently available metadata, but unit support will be incomplete and some corner cases may be missed.

CHAPTER 34

YTEP-0036: Converting from Nose to Pytest

34.1 Abstract

Created: September 30, 2019 Author: Jared Coughlin

This YTEP proposes two major changes to yt's answer testing:

- Switch from nose to pytest
- Store array hashes rather than full arrays

34.2 Status

In progress

34.3 Project Management Links

Relevant pull requests (chronological order from past to present):

- 2286
- 2401
- 2468
- 2548
- 2817
- 3102

34.4 Detailed Description

34.4.1 Background

Currently, testing in yt makes use of the nose framework. Issues with nose include:

- Being in a self-described maintenance mode for the last several years
- Lacking modularity
- Using lots of boilerplate code

The first proposal of this YTEP is to switch yt's testing framework from nose to pytest. Pytest offers many of the same benefits of nose:

- Automatic test discovery
- Ability to selectively run tests
- A large number of external plugins
- Fine-tuning via configuration files
- Compatibility with python's standard library testing framework unittest.

In addition to these benefits, pytest is also:

- Actively maintained and developed
- Compatible with nose
- Equipped with a fully-featured fixture system

In fact, this fixture system is arguably the best reason to use pytest. Benefits include:

- Greatly increases modularity
- Reduces boilerplate
- Makes writing tests easier
- Allows for smarter resource use when collecting tests

The second proposal of this YTEP is a change to the way answer test results are saved. Currently, many answer tests in yt generate large arrays of data that need to be saved in order to facilitate comparison with future test runs. The size of these arrays:

- Slows down answer comparison
- Necessitates that they be stored separately from the main yt code base, which serves to complicate answer comparison
- Synchronizing pull-request merging with two repositories instead of one also slows down the development itself and creates technical debt

In an effort to combat these issues, this YTEP proposes saving the hashes of the answer arrays. Since these hashes are short, simple strings, they:

- Can be stored in human-readable yaml files
- Take up much less disk space
- Facilitate more efficient comparisons
- Can be packaged with the code itself

34.4.2 Converting to Pytest

There are two steps for converting from nose to pytest:

- Rewrite each nose test class as a function
- Rewrite each answer test to employ pytest fixtures

Rewrite Nose Test Classes As Functions

Currently, the abstract answer tests are implemented as classes that use `yield` statements (e.g., `FieldValuesTest`). Pytest does not support yield tests due to conflicts with the fixture system.

As such, each nose test class' `run()` method is now a function named after the test class (e.g., `FieldValuesTest` becomes `field_values_test` and contains the code from the former's `run()` method). These abstract answer test functions are now contained in the following file: `yt/utilities/answer_testing/answer_tests.py`.

Rewrite Each Answer Test To Employ Pytest Fixtures

The answer tests (e.g., those contained in `yt/frontends/enzo/tests/test_outputs.py`) are now, where applicable, parameterized using the `@pytest.mark.parametrize` decorator, which removes the need to loop over various parameter combinations and makes logging the results of individual parameter combinations easier.

Conftest Files

These are configuration files that are used by pytest in order to define custom fixtures for processes such as setup, teardown, parameterizing, and using temporary directories and files.

The primary `conftest.py` file resides in the root of the yt repository. It:

- Defines the command-line options
- Defines the fixtures used across each of the answer tests

Testing the Tests

The pytest ecosystem contains a swath of useful tools that can be employed in order to aid the testing process. Several such tools are listed here:

- pytest-randomly is a plugin for causing the tests to be collected in a random order each time they are run. This helps guard against nefarious bugs that may result from calling tests in a specific order
- pytest-cov is a plugin that generates test coverage reports. It also plays well with other useful pytest plugins such as pytest-xdist, which allows for tests to be run in parallel
- coverage-badge is a plugin for generating a test coverage badge that can be added to the *README* file

Doctest Integration

In addition to being able to run both the unit and answer tests for yt, pytest can also run doctests embedded in documentation as well as source code doc string via the `--doctest-glob="*.rst"` command-line option, which is described here, and the `doctest_namespace` fixture, which is described **'here < https://docs.pytest.org/en/stable/doctest.html#doctest-namespace-fixture>'_**.

34.4.3 Saving Answer Test Results As Hashes

This is handled by the `hashing` fixture defined in the central `conftest.py` file. This fixture is then applied to every test that needs to save a result. The fixture applies the `md5` method of the `hashlib` library to get the hex digest of the arrays produced by the tests. Once completed, the hashes and test parameters are written to yaml files with the following format:

```
calling_function_name:
  test_name: hash
  test_parameter1: value
  test_parameter2: value
```

This produces human-readable text files that can be easily packaged with the main code base, which facilitates easier test management.

34.4.4 Running the Tests

The unit and answer tests are mutually exclusive, being run with two separate commands.

Similar to how the unit tests were run with nose, they can be run with

```
$ pytest
```

from the root yt repository directory.

To run a specific test or group of tests, one can either pass in the path to the module containing the tests

```
$ pytest /path/to/tests/test_module.py
```

or use pytest's `-k` flag, which enables test selection by name. For example, to run all of the tests contained in a single class, one would do:

```
$ pytest -k "TestClass"
```

To run only a specific method within a given class, one would do:

```
$ pytest -k "TestClass and test_method"
```

See this link for more on pytest's selection capabilities and options.

The first step is to tell `yt` where the test data is located

```
$ yt config set yt test_data_dir /path/to/yt-data
```

To run the answer tests for a specific frontend (e.g., tipsy)

```
$ pytest --with-answer-testing --answer-store -k "TestTipsy"
```

By default, the answers are stored in the location specified in `pytest_answer.ini`. This can be overridden from the command line

```
$ pytest --with-answer-testing --answer-store --local-dir=/path/to/save -k "TestTipsy"
```

Should one desire to save the actual arrays produced by the answer tests, this can be done with the following command line options

```
$ pytest --with-answer-testing --answer-raw-arrays --raw-answer-store
```

If the `--raw-answer-store` option is left off, then pytest will attempt to load in a set of previously generated arrays and perform a comparison to those generated during the current run.

34.4.5 Writing New Tests

Within the file containing the answer tests, one should define a new class that is marked by pytest as being an answer test. If the tests need to save data, they should utilize the `hashing` fixture. Additionally, if possible, the arguments passed to the test function should be parameterized. For example:

```
import pytest

dsList = [some_dataset, other_dataset]
param1List = [value1, value2]
param2List = [value1, value2]


@pytest.mark.answer_test
class TestNewFrontend:
    answer_file = None
    saved_hashes = None

    @pytest.mark.usefixtures("hashing")
    @pytest.mark.parametrize("ds", dsList, indirect=True)
    @pytest.mark.parametrize("param1", param1List, indirect=True)
    @pytest.mark.parametrize("param2", param2List, indirect=True)
    def test_method1(self, ds, param1, param2):
        test_result = some_answer_test(ds, param1, param2)
        self.hashes.update({"some_answer_test": test_result})
```

If desired, test parameterization can be handled in a `conftest.py` file that lives in the new frontend's `tests` directory. See the pytest documentation for more.

34.5 Community

The primary method of reaching out to the community about these changes is through the yt-dev mailing list.

These solutions will be tested by making sure that all of the current answer tests produce results that match those currently produced by nose.

34.6 Backwards Compatibility

This YTEP breaks backward compatibility of testing because testing will no longer be able to be done by nose.

CHAPTER 35

YTEP-0037: Code styling

35.1 Abstract

Created: May 18, 2020 Author: Clément Robert

This YTEP proposes the enforcement of code styling guidelines with auto-formatting tools.

35.2 Status

Completed

35.3 Project Management Links

The following PR are part of this proposal

- sorting imports with isort (#2592)
- code formatting with black (#2596)
- add a `pyproject.toml` file (#2598)
- add a precommit hook configuration file (#2600)
- add `flake8-bugbear` to our CI and fix existing errors (#2667 #2668 #2669 #2670 #2671 #2672 #2673 #2674)

And those are post-process PRs

- #2750 (merged)
- #2756 (merged)
- #2759 (merged)

- #2758 (merged)
- #2777 (merged)
- #2789 (merged)

35.4 Detailed Description

Code styling guidelines are already presented in the project's documentation, though enforcing them is not explicitly made part of the reviewing process.

We already use `flake8` and integrate it to our CI to catch a subset of infractions to PEP 8. From `flake8`'s pypi page

> **Flake8 is a wrapper around these tools:**
>
> - PyFlakes
> - pycodestyle
> - Ned Batchelder's McCabe script

From `black`'s documentation

> The rules for horizontal whitespace can be summarized as: do whatever makes pycodestyle happy. (. . .)
> The coding style used by Black can be viewed as a strict subset of PEP 8.

so it is expected that `black` plays nicely with `flake8` by construction. `black` applies an opinionated style, and offers very little configuration options by design. Only the target line length can be changed. This makes it a critical point, requiring discussion if this YTEP is approved.

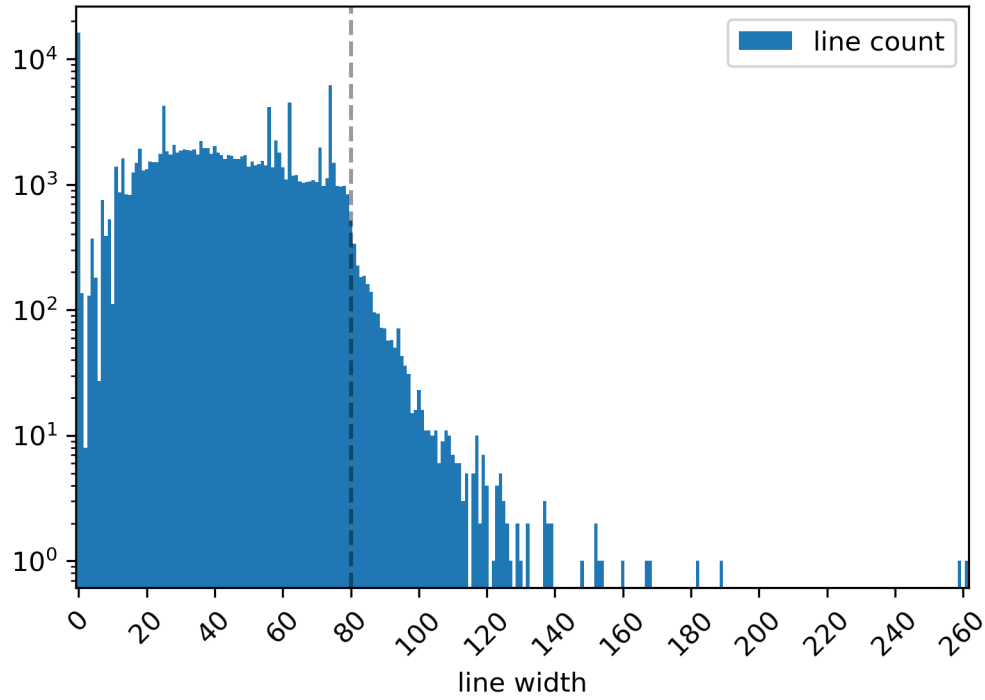
35.4.1 Maximal line length

Note: After discussion a maximum-line-length of 88, which is `black`'s default setting, was adopted.

The guidelines states that

> Line widths should not be more than 80 characters

Despite this being respected in most of the code base, there remains a large amount of outliers, that would be time-consuming to go through by hand. Taking the example of the yt-4.0 branch at the time of writing, there are 2158 lines exceeding 80 characters (~1.5% of the whole code base), or, visually



Note that when long strings are present, `black` will not attempt to split them to shorten the individual lines. This is most important in the case of dosctrings, and I explore the tools available to validate them hereafter (see *additional rules*).

There is a range of possible values we might give preference to. Python's standard library caps line-length at 79, pandas does so at 88. By default, `black` will target 88, as its authors claim it reduces the total number of lines by some 10% (as compared to enforcing 80).

In first drafting the PR linked above, I chose a line-length of 100, so as to minimize the amount of manual tweaking left to me after a `black` pass. I estimated that imposing a strict limit to 80 chars would leave 545 lines to be manually updated, while capping at 88 leaves a mere 135 (75% less work). As a reference, `dask` uses black's default settings, *and* allows flexibility for docstrings up to 120 characters through flake8.

35.4.2 Sorting imports

PEP8 recommends sorting imports statements, Needless to say, the task is daunting and definitely not worthy of anyone's time if we had to go back and apply those rules manually to the code base. Luckily, `isort` is able to check for and auto-apply those rules, so it can easily be added to the CI-linting process.

35.4.3 Additional rules & flake8 plugins

Since the oldest python version supported (as of yt 4.0dev) is 3.6, it means we can start using fstrings instead of `str.format()` and `%` formatting. #2605 demonstrates how a transition can be performed using `flynt`.

–

flake8-bugbear is a `flake8` plugging that goes beyond code style and detects some additional anti-patterns, most of which are correspond very likely to design flaws in otherwise syntactically valid statement. For instance, it will catch mutable default values such as in

```
def spam(a={}, b=[]):
    #...
```

which, in most contexts, should be rewritten as

```
def spam(a=None, b=None):
    if a is None:
        a = {}
    if b is None:
        b = []
```

This is a well known "gotcha", as documented for instance here. In short, this plugin detects bugs that went under the radar up to now, so it's probably worth adding it to our linting CI.

–

Another plugging can be added to enforce docstring formatting (flake8-docstrings), and has a straight-forward option configuration to validate docstrings are numpy-styled. However, there is currently a very large debt in errors caught by this tool, and no way to automatically solve them. However, it could still be added to our linting CI, if check for *new* errors only, such as

```
git diff upstream/master -u -- "*.py" | flake8 --diff
```

(snippet borrowed from pandas' contributing guide)

35.4.4 Side effects

Although some default options in `isort` conflict with `black`'s opinionated standard, it can be configured so that the tools play nicely with each other. This is demonstrated in #2596 where both check pass on Travis.

On another note, black only recognizes `pyproject.toml` as a configuration file (and is explicitly not planning to support other files such as `setup.cfg`). An undesirable effect of using `pyproject.toml` solely as a configuration file for `black` is that `pip` will detect it and change its behaviour when its present. The correct way to introduce this file is by specifying yt's build requirements within it. A proof of concept for this is #2598, where CI builds are run correctly across all tested python versions (3.6, 3.7, 3.8).

A serious counter-argument to applying black is that it implies messing up with `git blame` by making a single contributor the de facto last-author of a large number of lines they have not even necessarily read. Most recent versions of `git` can be configured to ignore specific commits in `git blame`. However, `black`'s own README currently points out that GitHub's UI for `git-blame` does not support this feature (yet ?).

It should be noted that `black` does not have a parser for Cython files, but interestingly `flake8` and `isort` do. Thus it is possible to add style checks for Cython extensions to the CI pipeline.

Additionally, `black` will not force line-length limits in docstrings. `flake8` will still be able to catch violations there, but solving them require manual tweaking. However, the amount of existing docstrings going over 88 characters is fairly small (a few dozens), so this is by no means a blocking condition.

35.4.5 Outreach and transition

Enforcing these change throughout future contributions can be done by

- updating the Developer Guide (done in part in #2592)
- offering a precommit hook configuration file to help contributors automate the linting stage locally (`precommit_hook.yaml`) such a configuration file is proposed in #2600

It is expected that transitioning to the "blackened" version of the code will add a bit of overhead in merging pre-existing PRs. Specifically, a simple `git merge <pr-branch> master` will almost certainly raise git conflicts. One possible solution to this is to sanitize the pr-branch (on author side) with:

```
pip install lint_requirements.txt
black yt/
isort .
git merge --strategy ours master
git push
```

I tested this strategy locally by simulating blackening at an arbitrary point in the past and merging the current state of the code base back in, producing a net zero diff with a direct blackening of the current state. In practice I advise caution, and sanitized code should be reviewed before merging. Another, arguably cleaner way to to resolve conflicts is to rebase the branch onto master and solve conflicts along the process. This is the prefered method though I would not recommend it to contributors who are not used to rebasing since it is easy to make mistakes in the process.

The shorter the transition, the easier, so I think that most of the PRs could be merged in a very narrow time window (a day or two), provided the appropriate conditions. However, because we want to ensure that each step passes the tests, which typically takes a least an hour or two per step, I propose that prep steps be done separately, and the big one (blackening) happen on a meeting.

35.5 Roadmap

To ensure cohesion in getting the number of features included in this PR in the codebase, we will have a dedicated maintainer/triage meeting. This YTEP's PR, the yt slack, and the yt-dev mailing list will include the meeting details for interested parties to attend. Some items require completion before the triage meeting, and some can be done afterwards, and have been categorized below.

35.5.1 Pre meeting

The following questions should be resolved * settle on a maximal line length (final: 88 characters) * decide on where should `unyt` import statement lands (for isort sorting): on third party section, or a custom intermediate section between first and third parties ? (final: third party)

35.5.2 On the meeting

- merge isort pass on the code base + CI check + doc (done)
- merge (needs tweaking) #2598 (done)
- merge blackening + manual fixups + CI checks + doc (done)
- signal to open PR authors that they should apply black (see transitioning strategy)

35.5.3 Can be done later

- merge #2600 (done)
- merge #2595 (done)
- reduce flake8 ignore list (done)
- add bugbear plugin and correct detected anti-patterns

35.6 Backwards Compatibility

Yes.

35.7 Alternatives

- Enforcing styling guidelines through peer review for each PR. Obviously this is a lot more work. Additionally, this methodology is prone to error and may cause delay in the PR approval process in case the authors disagree with the reviewers on the application of styling rules.
- Leaving code style decisions up to authors, and embracing the style diversity.

CHAPTER 36

YTEP-0039: Rich Terminal User Interface

36.1 Abstract

Created: March 3, 2021 Author: Clément Robert

Use `rich` to prettify our TUI (Terminal User Interface). Most notably logs, and progress bars (as a replacement for `tqdm`).

36.2 Status

Withdrawn

36.3 Project Management Links

- rich logging #3106
- rich progress bars #3114
- upstream, add support for non-tqdm based progress bars to pooch

  (unreleased as of May the 8th, 2021)

36.4 Detailed Description

rich is a library to build colorful and styled terminal user interfaces.

36.4.1 Logging

In particular it offers a `rich.logging.RichHandler` class that can be used to replace standard `logging.Handler` instances, such as the one currently used by yt.

It supersedes our custom code to turn on colors in log entries and overall produces much prettier (as well as more useful) logs at a marginal maintainance cost, arguably cheaper than our existing facility.

Let's illustrate our existing logger outputs and what `rich` turns them into, using the following script, and a minimalist configuration

```
import yt

yt.set_log_level(10)
yt.mylog.debug("2 + 2 = 5")
yt.mylog.info("Oh, people can come up with statistics to prove anything, Kent. 14% of␣
↪people know that.")
yt.mylog.warning("Don't eat that yellow snow.")
yt.mylog.error("I am Bender, please insert girder.")
yt.mylog.critical("You have 24h left to live.")

data = {
    "Nonetype": None,
    "a number": 1.657,
    "a boolean": True,
    "a list": [1, 2, 3, "spam", "bacon"],
}
yt.mylog.info("Logging some data:\n %s", data)
```

```
yt : [DEBUG    ] 2021-03-03 16:47:40,279 Set log level to 10
yt : [DEBUG    ] 2021-03-03 16:47:40,279 2 + 2 = 5
yt : [INFO     ] 2021-03-03 16:47:40,279 Oh, people can come up with statistics to prove anything, Kent. 14% of people know that.
yt : [WARNING  ] 2021-03-03 16:47:40,279 Don't eat that yellow snow.
yt : [ERROR    ] 2021-03-03 16:47:40,279 I am Bender, please insert girder.
yt : [CRITICAL ] 2021-03-03 16:47:40,279 You have 24h left to live.
yt : [INFO     ] 2021-03-03 16:47:40,279 Logging some data:
 {'Nonetype': None, 'a number': 1.657, 'a boolean': True, 'a list': [1, 2, 3, 'spam', 'bacon']}
```

```
[03/03/21 16:48:26] DEBUG    yt : Set log level to 10                                                    logger.py:37
                    DEBUG    yt : 2 + 2 = 5                                                          logging_demo.py:4
                    INFO     yt : Oh, people can come up with statistics to prove anything, Kent. 14%  logging_demo.py:5
                             of people know that.
                    WARNING  yt : Don't eat that yellow snow.                                        logging_demo.py:6
                    ERROR    yt : I am Bender, please insert girder.                                 logging_demo.py:7
                    CRITICAL yt : You have 24h left to live.                                         logging_demo.py:8
                    INFO     yt : Logging some data:                                                logging_demo.py:17
                              {'Nonetype': None, 'a number': 1.657, 'a boolean': True, 'a list': [1,
                             2, 3, 'spam', 'bacon']}
```

```
yt : [DEBUG    ] 2021-03-03 16:52:14,025 Set log level to 10
yt : [DEBUG    ] 2021-03-03 16:52:14,025 2 + 2 = 5
yt : [INFO     ] 2021-03-03 16:52:14,026 Oh, people can come up with statistics to prove anything, Kent. 14% of people know that.
yt : [WARNING  ] 2021-03-03 16:52:14,027 Don't eat that yellow snow.
yt : [ERROR    ] 2021-03-03 16:52:14,027 I am Bender, please insert girder.
yt : [CRITICAL ] 2021-03-03 16:52:14,028 You have 24h left to live.
yt : [INFO     ] 2021-03-03 16:52:14,028 Logging some data:
 {'Nonetype': None, 'a number': 1.657, 'a boolean': True, 'a list': [1, 2, 3, 'spam', 'bacon']}
```

```
[03/03/21 16:51:35] DEBUG    yt : Set log level to 10                                                    logger.py:37
                    DEBUG    yt : 2 + 2 = 5                                                          logging_demo.py:4
                    INFO     yt : Oh, people can come up with statistics to prove anything, Kent. 14%  logging_demo.py:5
                             of people know that.
                    WARNING  yt : Don't eat that yellow snow.                                        logging_demo.py:6
                    ERROR    yt : I am Bender, please insert girder.                                 logging_demo.py:7
                    CRITICAL yt : You have 24h left to live.                                         logging_demo.py:8
                    INFO     yt : Logging some data:                                                logging_demo.py:17
                              {'Nonetype': None, 'a number': 1.657, 'a boolean': True, 'a list': [1,
                             2, 3, 'spam', 'bacon']}
```

Note that `rich` adds a clickable path to the source file where each entry was emmited from. Only the filename + line number are displayed but those are actually absolute links. Advanced terminal apps like iterm support link integration to make the best out of `rich` logging.

`rich` is flexible, supports a handful of color systems, and adapts to the system it runs against, which makes it more robust than our existing on/off switch for colored logs.

Currently, colored logs are turned off by default and activated from yt's config file as

```
[yt]
colored_logs = true
```

With `rich`, colored logs could be turned on by default at no cost, and with no risk of crashing a shell lacking color support. Because `rich` offers a lot of configuration options, we could choose to expose some of them in yt's config file within a new, dedicated section, which I'm drafting here with proposed default values. This should be aligned with the current state of the documentation in #3106

```
[logging]

# replaces yt.log_level
# logging level can be specifies as case (insensitive) string
# and passed down to yt.utilities.logger.set_log_level
level = "INFO"

# replaces yt.colored_logs
use_color = false

# replaces yt.stdout_stream_logging as well as yt.suppress_stream_logging
# accepted values are "stderr", "stdout" and "none" (completely disable logging)
# this is case insensitive to avoid breakage if a user was to write e.g., "None"
stream = "stderr"

# this is passed to a logging.Formatter instance
format = "%(message)s"

# this is arguably a more sensible default that the legacy format (unspecified)
# where miliseconds are displayed.
# This default value mimicks rich's, but exposing it makes it more obvious how it
# can be customized.
date_format = "[%m/%d/%Y %H:%M:%S]"

# the other option would be "legacy", see Backwards Compatibility section below
handler = "rich"

# the following options are silently ignored when `handler = "legacy"`

# width <=0 leaves rich's default Console width (auto-adjusted if the window changes
# size)
# otherwise must be >0 and is the total size (in columns) of a log entry
width = -1

# path to a custom rich config file, either absolute or relative to the cwd
# this parameter should be a (non-empty) string when it's used.
custom_theme = ""
```

Note that it is pretty hard to come up with satisfying and intuitive solution to interpret a relative path for the `custom_theme` option. It could be interpreted by humans as relative to any of the following - the global config

file - the local config file - the current config file - the current working directory - the python script being run - yt's install dir (less likely)

For this reason, I am not convinced it's worth supporting relative paths at all, or exposing this option, but I'm willingly leaving to it in this state as the most experimental part of the (logging) project. Feedback will be collected to decide how it should or shouldn't work according to early addopters if any.

36.4.2 Progress bars and status

The `rich.progress` module offers progress bars that are arguably much cooler-looking than the leading concurent (and current yt dependency) `tqdm`. More importantly, they are also much more flexible in a multi-tasking context (threading). Typically, `rich` can display more than one progress bar at once without interrupting the logging stream. For a demo of this, run

```
python -m rich.progress
```

Coexisting progress bars open the possibility for mutli-tasking with long-running tasks in yt without sacrifying expressivity in logs and other outputs.

Note that rich also borrows so-called "spinners" from cli-spinners, which offer a nice alternative to progress bars to express on-going progress, in particular in tasks where completion time may be difficult to estimate. Try them for yourself with

```
python -m rich.spinner
```

Known caveats

- Progress bars + Jupyter lab bug: https://github.com/willmcgugan/rich/issues/830
- Progress bars would be defacto heterogenous with `pooch` (used for `yt.load_sample`)

because it only knows `tqdm`. Replacing `tqdm` or more realistically adding support for `rich.progress`, or even arbitrary progress bar classes in `pooch` clearly requires a change upstream and is not a high priority, but eventually, this looks feasible.

36.4.3 yt CLI

A marginal side effect is that interactive command line applications could be writen in simpler ways than with vanilla Python using `rich`.

For instance, let's look at a snippet that was proposed for inclusion our config migration script `yt config migrate` (see #3044)

```
prompt = "Perform the migration now [yn]? "
user_input = input(prompt).lower()
while user_input not in ("y", "yes", "n", "no"):
    print(f"Did not understand your input '{user_input}'. Please enter 'y' or 'n'.")
    user_input = input(prompt).lower()
if user_input in ("y", "yes"):
    migrate_config()
else:
    raise SystemExit("Migration not performed: exiting.")
```

This can be expressed much more efficiently using `rich.prompt`

```
from rich.prompt import Confirm
if not Confirm.ask("Perform the migration now ?"):
    raise SystemExit("Migration not performed: exiting.")
migrate_config()
```

At the time of writing however there is no clear spot where this functionality would shine in yt.

36.4.4 Testing

YTEP-0035 (pytest) is making progress and closing final implementation. `pytest` has builtin fixtures to capture logs (`caplog`) and standard outputs (`capsys`) to inspect them, which makes testing of logging format much easier.

I have started a branch to test and fix existing and new problems with the migration CLI `yt config migrate` in #3112, which relies on pytest.

36.4.5 Outreaching

- Release notes.
- config file migration/conversion facility: produce warnings when deprecated log-related parameters are found and offer to auto-convert them in place. In case a conflict is detected at runtime between old and new parameters, use the new ones, but advise the user to manually remove old ones (list them).

36.5 Backwards Compatibility

Downstream projects may rely on yt's existing logging format. Some users may also simply prefer this style over what rich offers. Even if we make `rich`'s logging handler the default, we could offer a option to restore the "legacy" logging format in yt's configuration file.

```
[yt.logging]
handler = "legacy"
use_color = false
```

Note that by construction, switching back to the legacy format would be an opt-in, which should be ok as long as it is properly documented in release notes. Keeping support for old-style progress bars would be relatively straightforward but it would create friction on the side of dependency specifications: if we support both styles at any point, then we have no way to formally specify `tqdm` OR `rich` is required but not both. Considering this, I suggest to simply drop `tqdm` and make `rich` a hard dependency immediately (#3114).

Functions `yt.utilities.logging.colorize_logging` and `yt.utilities.logging.uncolorize_logging` won't be necessary anymore except if we want to maintain full backwards compatibility (legacy handler + color). What should be done with them is up for discussion but here's my personnal opinion. They live in a pretty nested part of the yt namespace, but they may be used downstream since they are not explicitly (or implicitly) documented as private. I think it's unlikely that anyone would want to use them at runtime instead of configuring yt, so this backwards compatibility breakage is likely acceptable. I also can't think of a reasonable workflow for which users would care about pretty logs *and* wish to keep the legacy format at the same time. I propose to mark them as deprecated until the next release (following the acceptance of this YTEP), then remove them.

36.6 Cost

`rich` would be added to yt's requirements.

36.7 Alternatives

Keep simple logs + `tqdm`.

CHAPTER 37

YTEP-0040: a yt-baked colormap package

37.1 Abstract

Created: 20 April, 2021 (happy 40th birthday Matt !) Author: Clément Robert

I propose to extract native colormaps (“cmaps” hereafter) from yt into a separate, lightweight package.

37.2 Status

Completed

37.3 Project Management Links

PRs following the release of Matplotlib 3.4

- remove duplicated (vendored) cmaps from matpltolib (cubehelix in #3149 and turbo in #3137)
- stop registering colormaps from cmocean #3175
- add a visible deprecation warning for unprefixed cmocean maps (#3214)
- document the deprecation above #3207
- bugfix an undesirable side effect from #3175 #3212

GH issue with the initial discussion #3165

The new package’s home is https://github.com/yt-project/cmyt

37.4 Detailed Description

An undocumented behaviour change in Matplotlib 3.4.0 made naming collisions in cmaps fatal. Registering a new cmap with a previously registered name will not work any more, and cause the program to crash. Up to yt 3.6, we've been automatically registering cmaps from various sources. Most notably: - yt itself - IDL - cmocean

All of the cmaps in questions are (or used to be) registered with bare names, i.e. without a dedicated prefix. This means that if any of our cmaps' names is used in the future release of Matplotlib, importing yt will fail with a `ValueError`.

Good practice is in fact to register cmaps under a custom "namespace" (prefix) that is guaranted to never collide with matplotlib native colormpas. This is what is currently done in cmocean itself (prefixing with `"cmo."`) as well as, for instance cmasher (prefixing with `"cmr."`).

Since migrating away from the fragile current registration method requires a structural change anyway, I propose we seize the opportunity to make yt native cmaps easier to use from outside the framework (to footnote: it is actually already doable but it requires loading the entire package) and export them into a small, lightweight, easy to maintain and install package.

Such a package would be a hard dependency of yt itself.

In case this proposal is accepted, the exact prefix used is up for discussion. I can propose 1) `cmy.` pros: perfectly fits the dominant convention, cons: doesn't look like yt 1) `yt.` pros: dead simple, best at "brand" reinforcement, cons: doesn't align with the dominant convention 1) `cmyt.` pros: has "yt" _and_ "cm" in the name, cons: one char more that the dominant convention

My personal preference goes to `cmyt.`, and I'm going with it as the proposed package name in the following.

NOTE: the fate of IDL orignated colormaps currently registered and vendored by yt is not clear. They could be ported as yet another package (and maybe made a hard dep to yt too ?), or be included in the first package (not my favourite option).

37.5 Backwards Compatibility

Forcing people to add prefixes everywhere is not (yet ?) necessary, but the practice should be encouraged nevertheless. One way to bridge the gap between current "malpractice" and a more stable usage in yt would be to perfom reregistrations of `cmyt`'s cmaps without a prefix (similarly to what was previously done with cmocean, though in an error-safe way).

Maybe this could be done silently in yt 4.0 (to save our users as much of a migration burden as possible), then raise visible deprecation warnings starting from yt 4.1, and finally be completely deprecated in yt 4.2 or yt 4.3 .

37.6 Alternatives

- Perform structural changes in how yt registers its own cmaps (necessary) without making

the result a new package (not necessary, but offers some additional benefits).

- Perform the necessary changes on yt side, export the cmaps into a separate package but keep them in the main package too (code duplication, not my cup of tea).

CHAPTER 38

YTEP-1000: GitHub Migration

38.1 Abstract

Created: March 25, 2017 Author: Lots of folks

The primary source code and project management for yt should be moved from bitbucket to github, and as a result, from mercurial to git. This document outlines a timeline, conversion and import mechanism, and how to manage this transition.

38.2 Status

Completed

The yt steering committee has evaluated the broad strokes, and it was presented to the yt community at the beginning of March. There were no objections, and this document is to be iterated on to decide on the migration strategy and timeline.

38.3 Project Management Links

- Mailing list message describing the situation: http://lists.spacepope.org/pipermail/yt-dev-spacepope.org/2017-March/006792.html

38.4 Detailed Description

As discussed in the email to yt-dev, we should move from BitBucket to GitHub as a result of the pervasive network effect of GitHub and its impact on the community of yt developers.

The process of migration needs to be planned to minimize disruption to developer and user workflow. It will proceed in roughly three stages.

38.4.1 Stage 1

A clone is created on GitHub, and this is evaluated by the developers. The specifics of the conversion procedures (specifically, things like any branch filtering, tag conversions, embedding of hashes and so on) will be examined. A few particular items of note are that the changeset hashes in commit messages and issues are of particular interest. Any commit messsage filtering or file size reduction will be considered. This clone will not be final, and will not be open for pull requests, changes, etc. It will be a one-way and one-time sync.

Whatever methods are used to do this conversion will be written in this YTEP.

An example using *fast-export* can be found at http://github.com/yt-project/yt_fast_export . This includes *git notes* for each changeset hash, but they must be pulled specifically using *git fetch origin "refs/notes/*:refs/notes/*"* .

Possible timeline: Starting April 10, 2017

To Do items *during* this stage:

- Create initial clone (this can be done manually)
- Test migration of issues to GitHub (1:1 mapping of numbers)
- Set up Jenkins jobs using git and GitHub plugins
- Rewrite *yt update*
- Update documentation
- Update slack bot
- Migrate supplemental repositories to GitHub

38.4.2 Stage 2

This is a brief stage, during which we wind down PRs from the Bitbucket repository. All PRs will be either accepted or declined. All PRs that are declined that are still "in progress" will need to be converted to github pull requests, the process for which will be documented. The simplest mechanism will be through *hg export* and *git import*, which will squash patches.

Possible timeline: Starting May 1, 2017. This stage is contingent on the Stage 1 To-Do's being completed, but should be roughly three weeks after Stage 1 is entered.

To Do items *during* this stage:

- Accept or decline all PRs on BitBucket. Those PRs that are not accepted by the conclusion will need to be moved to GitHub
- Switch infrastructure over to GitHub
- Open up PRs on GitHub and begin code review there
- Migrate users (ask for their GH handle) and their access levels

38.4.3 Stage 3

This is the final stage. At this point, the switch will be flipped, and no more PRs or code review will be accepted on BitBucket.

Possible timeline: Starting May 15, 2017. This stage is contingent on the Stage 2 To-Do's being completed, but should be roughly two weeks after Stage 2 is entered.

To Do items *during* this stage:

- None that I can think of.

Once Stage 3 has been completed, the bitbucket repository will be marked read-only.

38.4.4 Progress and Notes

During the course of the stages, we will update this YTEP with notes on conversion processes, etc.

38.5 Backwards Compatibility

This will almost certainly *not* break internal APIs other than those identified in the "todo" section above, which are all related to project maintenance like updating and so on.

The developer workflow will break, but we are attempting to mitigate that through this YTEP.

Finally, the documentation (identified as needing to be updated) will be updated to reflect the new normal.

CHAPTER 39

YTEP-3000: Let's all start using yt 3.0!

39.1 Abstract

Created: October 30, 2013 Author: Matthew Turk

This is a YTEP suggesting we all start using yt 3.0 for development, and where the blockers to adoption are enumerated.

39.2 Status

Completed

39.3 Project Management Links

Basically all the project management links up to this point have been talking about this.

39.4 Detailed Description

This YTEP outlines the items necessary to be implemented before yt 3.0 can be released, and before we can attempt to move development and day-to-day usage for developers to the 3.0 codebase.

There are essentially three categories of work items: release blockers, necessary features to migrate usage and development, and feature parity requirements.

Several developers have expressed that a major blocker is concluding work they have begun on yt 2.6 and the 2.x branch; this document is meant to supplement that, rather than replace it.

39.4.1 Release Blockers

These are components that need implementing before yt 3.0 can be released. This is not the same as reaching a "complete" implementation; the important work is to ensure that subsequent API breakages are minimal. We are tracking these on the yt-3.0 Trello Board.

- Merging unitrefactor; waiting only on documentation and rebranding merge at this time. (*YTEP-0011* and *YTEP-0017*.)
- De-astroification of yt and renaming of generic objects which has been mostly accomplished in the rebranding bookmark.
- Removing dict-like access to static output (*YTEP-0018*), not yet compelted in the rebranding bookmark.
- Considerable amount of documentation, which is being worked on.

I do not believe there are any other blockers to yt 3.0.

39.4.2 Necessary Features

These are items that are necessary for developers to migrate from using and developing yt 2.6 to yt 3.0.

This is intentionally left mostly empty, as items from "feature parity" will be migrated up.

- `field_cuts` (which is a related to `cut_region`, which has been mostly implemented.)

39.4.3 Feature Parity

These are items that existed in yt 2.6 that do not exist in yt 3.0 yet.

- A handful of hierarchy attributes have not yet been implemented.
- A few frontends still need polishing during the port, including Chombo, Pluto, NMSU-ART, and GDF. These are small items but will need assistance from individual frontend maintainers.
- The sidecar storage has not been ported.
- Boolean regions have not been implemented. They can likely be implemented in the same manner as `cut_region` has been.

39.5 Backwards Compatibility

This does not add any new backwards incompatible items, it is merely a call to action.

39.6 Alternatives

Call a mulligan, start over?

CHAPTER 40

YTEP-9999: YTEP Template

To write a YTEP, copy this template to the next numerical number, add it to the repository, and issue a pull request. Discussion of the YTEP will occur either on the mailing list (for large-scale changes) or in the PR itself (small items, such as formatting).

This document has been patterned after the Matplotlib Enhancement Proposal Template (found here.)

40.1 Abstract

Created: November 25, 2012 Author: Your Name

This section should contain one or two sentences describing the proposed change. It should not contain detailed design information, but it can contain background information.

This should contain a date

40.2 Status

Status should be one of the following:

1. Proposed
2. Completed
3. In Progress
4. Declined

YTEPs do not need to pass through every stage.

40.3 Project Management Links

Any external links to:

- Pull requests
- Related issues in the bug tracker
- Previous implementations
- Mailing list discussions or Google Docs

40.4 Detailed Description

Here is where you should write detailed description of what the YTEP proposes. This needs to include:

- Background
- Nature of the problem
- Nature of the solution
- How will the solution be implemented * Brief outline of the code needed to implement this * Code examples of using the solution, in appropriate * How will the solution be tested?
- What are any stumbling points
- What is the proposed method for reaching out to the community about this?

40.5 Backwards Compatibility

This section should outline backwards compatibility issues. In particular, it should focus on those issues that will appear to the main scripting API: will this break old scripts? Will it break internal uses of the API?

40.6 Alternatives

This section is optional.

What other means are there to accomplishing the goals of this YTEP, and why is this the best option?

Bibliography

[DA2012] Dehnen W., Aly H., 2012, MNRAS, 425, 1068

[PricePaper] http://adsabs.harvard.edu/abs/2012JCoPh.231..759P

[SPLASHPaper] http://adsabs.harvard.edu/doi/10.1071/AS07022