Introducing yt 3.0: Analysis and Visualization of Volumetric Data

This manuscript was automatically generated from yt-project/yt-3.0-paper@8accaad on February 15, 2018.

Authors

- **The yt Project**
  XXXX-XXXX-XXXX-XXXX · yt-project · yt_astro
  NumFOCUS · Funded by Grant XXXXXXXX

- **Matthew Turk**
  0000-0002-5294-0198 · MatthewTurk · powersoffour
  School of Information Sciences, University of Illinois at Urbana-Champaign; Department of Astronomy, University of Illinois at Urbana-Champaign · Funded by Grant XXXXXXXX

- **Add Yourself**
  XXXX-XXXX-XXXX-XXXX · yournamehere
  Department of Something, University of Whatever; Department of Whatever, University of Something

- **Jane Roe**
  XXXX-XXXX-XXXX-XXXX · janeroe
  Department of Something, University of Whatever; Department of Whatever, University of Something

Abstract

Abstract here!

Introduction

The process of transforming data into understanding constitutes the vast majority of time, energy, and intellectual effort spent during scientific inquiry. This is true across domains, whether data is the product of a computational simulation, a telescope observation, the synthesis of sensors distributed across the Earth, or a collection of images of the human brain. Data, by themselves, do not reflect an understanding of the Universe or its underlying physical properties; rather, they are recordings, or measurements, of the state of systems as observed. Even for computational simulations, such as simulations of star formation in the galaxy, this is true: these simulations encode information about a discretization of a model, rather than the model itself.

Bridging the gap between this discretization and the physical understanding requires accessing data, manipulating and interrogating this data, and then applying to this data a sense of understanding. Somehow, bits stored on a disk must become, in our minds, a galaxy undergoing a starburst.

This process is both mediated and impeded by computational tools. When those tools align with our mental model of how data exists, they can allow us to work more efficiently, asking questions of data and building sophisticated scientific inquiry. However, when they do not, they can cause frustration, delays, and most worryingly, incorrect or misinterpreted results. When viewing this from the perspective of the landscape of inquiry, the most startling realization is that the questions a computational tool enables individuals to ask shapes the questions they think to ask.

In [1], the analysis platform `yt` was described. At the time, `yt` was focused on analyzing and visualizing the output of grid-based adaptive mesh refinement hydrodynamic simulations; while these were used to study many different physical phenomena, they all were laid out in roughly the same way, in rectilinear meshes of data. In this paper, we present the current version of `yt`, which enables identical scripts to analyze and visualize data stored as rectilinear grids as before, but additionally particle or discrete data, octree-based data, and data stored as unstructured meshes. This has been the result of a large-scale effort to rewrite the underlying machinery within `yt` for accessing data, indexing that data, and providing it in efficient ways to higher-level routines, as discussed in Section Something. While this was underway, `yt` has also been considerably reinstrumented with metadata-aware array infrastructure, the volume rendering infrastructure has been rewritten to be more user-friendly and capable, and support for non-Cartesian geometries has been added.

The single biggest change to `yt` since that paper was published has not been technical in nature. In the intervening years, a directed and intense community-building effort has resulted in the contributions from over a hundred different individuals, many of them early-stage researchers, and a thriving community of both users and developers. This is the crowning achievement of development, as we have attempted to build `yt` into a tool that enables inquiry from a technical level as well as fosters a supportive, friendly community of individuals engaged in self-directed inquiry.

Community Building

Choosing a software package for a particular purpose involves evaluating several differentiating factors; these factors include the functionality of a package, the performance of a package, the user-friendliness, and even the ability of an individual to find help, engage with others and feel a sense of participation. **cite something here** The development, fostering and design of the community around `yt` is deemed to be both crucial to the success or failure of `yt`, and in many ways inseparable from its functionality.

Composition

There are several rough categories of individuals engaged in development and utilization of `yt`. As a result of its API-first design, there are few if any individuals who use `yt` that do not do so through the scripting interface; this means that the vast (if not exclusive) majority of individuals who interact with the functionality in `yt` are doing so by writing their own scripts, modules, and code, and arguably engaging in a value-added development process of their own. The majority of individuals using `yt` at present are in astronomy and astrophysics, typically fields of simulation, although there is an increasing group of individuals from other domains that are participating in development and using `yt` for their own domain-specific problems.

Making the distinction somewhat more clearly, there are individuals who have built their own scripts and utilized them as well as individuals who have contributed changes or modules to the primary `yt` codebase. In addition, there is an emerging set of projects that build on `yt` as infrastructure to conduct scientific analysis. These developers are largely driven by their own pragmatic scientific needs, and they constitute the majority of developers (by number) that contribute to the code base. The majority of these individuals are early- to mid-career researchers, typically graduate students, postdocs, and assistant professors.

In recent years, there has emerged a more coherent contingency of individuals who participate in both pragmatically-focused development of modules and functionality for their own benefit as well as modules or overall improvement that is supplemental or even external to their own research agenda. These improvements include improvements to the unit handling, to the plotting code, to

infrastructure for loading disparate datasets, and so on. At this time we do not know of any individuals funded to work on `yt` completely independent of a scientific or scholarly goal.

The composition of the community, particularly with a mixture of timelines for goal-setting and completion, can at times cause frustrations and difficulties. For instance, the response to “Can this feature be implemented?” often includes an invitation for the questioner to collaborate on developing that feature and submitting it to the codebase. Developing a schedule of releases is an act of consensus building, both deciding what bugs are critical to fix in the timeline of a release as well as building consensus on what features should be considered blockers for a new release. The intersection of this with academic deadlines (for instance job application season) requires balance and care.

Types of Tasks

When evaluating the level of engagement, we consider a few different classifications of tasks that are performed by individuals in the community, and evaluate these based on how they flow into greater engagement.

- Filing issues
- Participating in mailing list discussions
- Issuing a pull request
- Writing documentation
- Participating in code review
- Drafting an enhancement proposal
- Closing bug reports

While there are other activities that individuals can participate in, these are the typical activities we see among participants in the community. The order, flowing from the first to the last, is the typical flow we see for an individual coming to participate in the community. The first step is typically to file an issue or bug report (occasionally these are requests for new features), followed by partipating in development-focused discussion on mailing lists. The next level of engagement typically involves the development of a new piece of functionality, refinement of existing code, or issuing a fix for a bug or issue. These take the form of pull requests (described in greater detail here) that can be reviewed and added to the code base.

The next level of engagement centers around tasks that are not fully-aligned with pragmatic, code-driven scientific inquiry. The development of documentation is often viewed as orthogonal to the scientific process, and typically requires an iterative wrriting process. Participation in code review, providing comments, feedback and suggestions to other authors, is another somewhat orthogonal task; it doesn’t necessarily directly benefit the developer doing the reviewing (although it might) and it does not necessarily result in academic rewards (citations, authorship, etc). But, it does arise

from a pragmatic (ensuring code reliability) or altruistic (the public good of the software) motivation, and is thus a deeper level of engagement.

The final two activities, drafting enhancement proposals and closing bug reports, are the most engaged, and often the most removed from the academic motivation structure. Developing an enhancement proposal for `yt` means iterating with other developers on the motivation behind and implementation of a large piece of functionality; it requires both motivation to engage with the community and the patience to build consensus amongst stakeholders. Closing bug reports – and the development work associated with identifying, tracking and fixing bugs – requires patience and often repeated engagement with stakeholders.

Engagement Metrics

We include here plots of the level of engagement on mailing list discussions and the citation count of the original method paper.

Governance

Development Procedure

Answer Testing

Code Review

YTEP Process

Data Objects

The basic principles by which `yt` operates are built on the notion of selecting data (through coarse and subsequent fine-grained indexing of data sources such as files), accessing that data in a memory-efficient fashion, and then processing that data into either a resultant set of quantitative data or a visualization.

The mechanisms by which `yt` can select data are typically spatial in nature, although several non-spatial mechanisms focused on queries can be utilized as well. These objects which conduct selection are selectors, and are designed to provide as small of an API as possible, to enable ease of development and deployment of new selectors.

Selectors require defining several functions, with the option of defining additional functions for optimization, that return true or false whether a given point is or is not included in the selected

region. These functions include selection of a rectilinear grid (or any point within that grid), selection of a point with zero extent and selection of a point with a non-zero spherical radius.

The base selector object utilizes these routines during a selection operation to maximize the amount of code reused between particle, patch, and octree selection of data. These three types of data are selected through specific routines designed to minimize the number of times that the selection function must be called, as they can be quite expensive.

Selecting data from a grid is a two-step process. The first step is identifying which grids intersect a given data selector; this is done through a sequence of bounding box intersection checks. Within a given grid, the cells which are intersected are identified. This results in the selection routine being called once for each grid object in the simulation and once for each cell located within an intersecting grid. This can be conducted hierarchically, but due to implementation details around how the grid index is stored this is not yet cost effective.

Selecting data from an octree-organized dataset utilizes a recursive scheme that selects individual oct nodes, then for each cell within that oct, determining which cells must be selected or child nodes recursed into. This system is designed to allow for having leaf nodes of varying cells-per-side, for instance 1, 2, 4, 8, etc. However, the number of nodes is fixed at 8, with subdivision always occurring at the midplane.

The final mechanism by which data is selected is for discrete data points, typically particles in astrophysical simulations. At present, this is done by first identifying which data files intersect with a given selector, then selecting individual points. There is no hierarchical data selection conducted in this system, as we do not yet allow for re-ordering of data on disk or in-memory which would facilitate hierarchical selection through the use of operations such as morton indices.

add bitmap index stuff here

Processing and Analysis of Data

Abstracting Simulation Types

Chunking and Decomposition Strategies

Reading data, particularly data that will not be utilized in a computation, can incur susbtantial overhead, particularly if the data is spread over multiple files on a networked filesystem, where metadata queries can dominate the cost of IO. `yt` takes the approach of building a coarse-grained index based on the discretization method of the data (particle, grid, octree, unstructured mesh), combining this with datapoint-level indexing for selection processes.

To supplement this, methods in `yt` that process data utilize a system of data "chunking," whereby segments of data identified during coarse-grained indexing are subdivided by one of a few different schemes and yielded to the iterating function; these schemes can include a limited number of tuning parameters or arguments. These three chunking methods are `all`, `spatial` and `io`. The `all` method simply returns a single, one-dimensional array, and the number of chunks is always exactly one; this enables both non-parallel algorithms and simple access to small datasets. `spatial` chunking yields three-dimensional arrays. For grid-based datasets, these are the grids, while for particle and octree datasets they are leaf-by-leaf collections of particles or mesh values. Optionally, the `spatial` chunking method can return "ghost zones" around regions, for computation of stencils. The final type of chunking, `io`, is designed to iterate over sets of data in a manner that is most conducive to pipelined IO. These will not always be load-balanced in size of the returned chunks, however. In some cases, `io` chunking may return one file at a time (in the case of spreading items across many different files), while in others it may be returning sub-components of a single file. This chunking type is the most common strategy for parallel-decomposition.

Necessarily, both indexing and selection methods must be implemented to expose these different chunking interfaces; `yt` utilizes specific methods for each of the primary data types that it can access. We detail these below, specifically describing how they are implemented and how they can be improved in future iterations.

Grid Analysis

`yt` was originally written to support the Enzo code, which is a patch-based Adaptive Mesh Refinement (AMR) simulation platform. Analysis of grid-based data is the most frequent application of `yt`. While we discuss much of the techniques implemented for datasets consisting of multiple, potentially overlapping grids, `yt` also supports single-grid datasets (such as FITS cubes) and is able to decompose them for parallel analysis.

`yt` also supports other grid patch codes **insert list here**

`yt` supports several different "features" of patch-based codes. These include grids that span multiple parent objects, grids that overlap with coarser data (i.e., AMR), grids that overlap with other grids that provide the same level of resolution of data (i.e., grids at the same AMR level), refinement factors that vary based on level, and edge- and vertex-centered data. For the cases of overlapping grids (either on the same or higher refinement levels) masks are generated that indicate which data is considered authoritative.

As noted in Data Objects, the process of selecting points is multi-step, starting at coarse selection that may be at the file level, and proceeding to selection of specific data points that are included in a selector. For grid-based data, the coarse selection stage proceeds in an extremely simple fashion, by iterating over flat arrays of left and right grid edges and creating a bitmap of the

selected grids. Because this method – while not taking advantage of any data structures of even mild sophistication – is able to take advantage of pipelining and cache-optimization, we have found that it is sufficiently performant in most geometries up to approximately 10^6 grid objects. In those cases, the distinction between "wide and shallow" grid structures (where refinement occurs essentially everywhere, but not to a great degree) and "thin and deep" grid structures (where refinement occurs in essentially one location but to very high levels), as well as the specific selection process, impact the overall performance. The second-stage selection occurs within individual grids, where points are selected based on the data point center. In the case of cell-centered data, this returns an array of size N where N is the number of points selected; in the case of 3D vertex-centered data, this would be $(N, 8)$. **Andrew Myers: check this?**

Indexing grid data in `yt` is optimized for systems of grids that tend to have larger grid patches, rather than smaller; specifically, in `yt` each grid patch consists of a Python object, which adds a bit of overhead. In the limit of many more cells than grid objects, this overhead is small, but in cases where the number of grids is $O(10^7)$ this can become prohibitive. These cases are becoming more common even for medium-scale simulations.

To address both the memory overhead and the python overhead, as well as more generally address potential scalability issues with grid selection, we have begun implementation of a more sophisticated "grid visitors" indexing and selection method. This draws on the approach used by the oct-visitors (described below). A spatial tree is constructed, wherein parent/child relationships are established between grids. Each process – selection, copying of data, generation of coordinates – is represented by an instance of a `GridVisitor` object. The tree is recursively traversed, and for all selected points the object is called. This allows grids, their relationships, and the data masks to be stored in structures and forms that are both optimized and compressed. This method is essential for scaling to a large number of grid patches; the storage requirements of a single grid patch Python object are around 1K per object (about one gigabyte per million grids), whereas the optimized storage reduces this to approximately 140 bytes (about one gigabyte per eight million grids), with further reductions possible; for selection operations, we are also able to reduce the number of temporary arrays and utilize compressed mask representation, bringing peak memory usage down further. The spatial-tree optimization substantially increases performance for "wide and shallow" dataset selection.

Octree Analysis

SPH Analysis

Unstructured Mesh Analysis

Non-Cartesian Coordinates

Visualization and Volume Rendering

Units and Quantities

At a basic level, `yt` is an engine for converting data dumped to disk by a simulation code into a physically meaningful result. Attaching units to simulation data makes it possible to perform dimensional analysis on the simulation data, adding additional opportunities for catching errors in a data processing pipeline. In addition, it becomes straightforward to convert data from one unit system to another.

In `yt` 3.0 we handle units in a an automatic fashion, leveraging the symbolic math library `sympy`. Instead of returning a NumPy `ndarray` when users query `yt` data objects for fields, return a `YTArray`, a subclass of `ndarray`. `YTArray` preserves `ndarray`'s array operations, including deep and shallow copies, broadcasting, and views. Augmenting `ndarray`, `YTArray` attaches unit metadata to the array data, enabling runtime checking of unit consistency in arithmetic operations between `YTArray` instances, and making it trivial to compose new units using algebraic operations.

As a trivial example, when one queries a data object (here given the generic name `dd`) for the density field, we get back a YTArray, including both the simulation data for the density field, and the units of the density field, in this case g/cm^3:

```
>>> dd[`density']
YTArray([4.92e-31, 4.94e-31, 4.93e-31, ...,
         1.12e-25, 1.59e-25, 1.09e-24]) g/cm**3
```

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']
  [4.92e-31 4.94e-31 4.93e-31 ...
   1.12e-25 1.59e-25 1.09e-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. `YTQuantity` is a subclass of `YTArray` with the requirement that the `array data'' associated with the instance be 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
[ 1.28e-12 1.29e-12 1.29e-12 ...
  1.63e-12 1.59e-12 1.40e-12] erg
```

If a user needs the field in a different unit system, they can quickly convert using `convert_to_units` or `in_units`.

When a `Dataset` 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 registry includes both concrete physical units like `cm` or `K` but also units symbols that correspond to the unit system used internally in the simulation.

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 `Dataset` has a `length_unit`, `time_unit`, and `mass_unit`, attribute that the user can quickly and easily query to discover the base units of the simulation. For example:

```
>>> import yt
>>> ds = yt.load("Enzo_64/DD0043/data0043")
>>> print ds.length_unit
128 Mpccm/h
>>> print ds.quan(1.0, "code_length").in_units("Mpccm/h")
128 Mpccm/h
>>> print ds.length_unit.in_cgs()
5.55517285026e+26 cm
```

Optionally `velocity_unit`, `pressure_unit`, `temperature_unit`, and `density_unit` may be defined as well if the units for these fields cannot be inferred from the mass, length, and time units.

Additionally, we allow conversions to the simulation unit system. Data in code units are available by converting to `code_length`, `code_mass`, `code_time`, `code_velocity`, `code_density`, `code_magnetic`, `code_pressure`, `code_metallicity`, or any combination of those units. Code

units 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 are 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:

```
>>> import yt
>>> ds = yt.load(``Enzo_64''/DD0043/data0043")
>>> dd = ds.all_data()
>>> print dd[('enzo', 'Density')]
[ 6.74e-02 6.12e-02 8.92e-02 ...
  9.09e+01 5.66e+01 4.27e+01] code_mass/code_length**3
>>> print dd[('gas', 'density')]
[ 1.92e-31 1.74e-31 2.54e-31 ...
  2.59e-28 1.61e-28 1.22e-28] g/cm**3
```

Here, the first data object query is returned in code units, while the second is returned in CGS units. This is because `("enzo", "Density")` is an on-disk field, while `("gas", "density")` is an internal `yt` field.

Implementation}

Our unit system has 6 base dimensions, `mass`, `length`, `time`, `temperature`, and `angle`. The unitless `dimensionless` dimension, which we use to represent scalars is also technically a base dimension, although a trivial one. For convenience, we also create dimensionless unit symbols to represent quantities like metallicity that are formally dimensionless, but it is convenient to represent in a unit system.

For each dimension, we choose a base unit. Our system's base units are grams, centimeters, seconds, Kelvin, 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. In the future, we plan to make the preferred internal coordinate system a user-configurable option.

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 an SI 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.

`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.

Creating YTArray and YTQuantity instances {sec:creating-ytarray-and-ytquantity-instances}

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 {sec: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, a 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 `Dataset` 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:

```
>>> import yt
>>> ds = yt.load(``Enzo_64/DD0043/data0043'')
>>> ds.arr([1, 2, 3], `code_length').in_cgs()
YTArray([ 5.55e+26, 1.11e+27, 1.66e+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 {sec: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 == 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
>>> 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
```

Handling 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 {sec: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 `Dataset` 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 `Dataset` instances.

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 they should also be defined in subclasses by extending this function.

Mixing modified unit registries {sec: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:

```
>>> import yt
>>> ds1 = yt.load(`Enzo_64/DD0002/data0002')
>>> ds2 = yt.load(`Enzo_64/DD0043/data0043')
>>> print ds1.length_unit, ds2.length_unit
128 Mpccm/h, 128 Mpccm/h
>>> print ds1.length_unit.in_cgs()
6.26145538088e+25 cm
>>> print ds2.length_unit.in_cgs()
5.55517285026e+26 cm


>>> print ds1.length_unit*ds2.length_unit
145359.100149 Mpccm**2
>>> print ds2.length_unit*ds1.length_unit
1846.7055432 Mpccm**2
```

For the last two examples, 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. Results are always consistent when referencing an unambiguous physical coordinate system:

```
>>> 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
```

Handling cosmological units {sec:handling-cosmological-units}

If we detect that we are loading a cosmological simulation performed in comoving coordinates, extra comoving units are added to the dataset's unit registry. Comoving length unit symbols are still named following the pattern `<length symbol>cm`, i.e. `Mpccm`.

The h symbol is treated as a base unit, `h`, which defaults to unity. The `Dataset.set_units` updates the `h` symbol to the correct value when loading a cosmological simulation.

User-Friendliness

Publication-Ready Figures}

IPython Integration and HTML GUIs

Halo-Finding and Catalogs

Scaling and Parallelism

Performance of Operations

Inline Analysis

Simple Parallelism

Analysis Modules

Future Directions

Conclusions

Acknowledgments

The authors of this paper would like to extend their deepest gratitude to the many, many individual and institutions that have contributed, directly or indirectly, to the growth of both `yt` and the `yt` community.

We particularly thank KIPAC and SLAC at Stanford, the University of California at San Diego and Santa Cruz, the High-Performance Astro Computing Center, Columbia University, the University of Illinois, University of Colorado at Boulder, University of Edinburgh, the scientific python community, NumFOCUS,

References

1. **yt: A MULTI-CODE ANALYSIS TOOLKIT FOR ASTROPHYSICAL SIMULATION DATA**
Matthew J. Turk, Britton D. Smith, Jeffrey S. Oishi, Stephen Skory, Samuel W. Skillman, Tom Abel, Michael L. Norman
The Astrophysical Journal Supplement Series (2010-12-28) https://doi.org/10.1088/0067-0049/192/1/9