September 2008
SLAC-PUB-13416

Analysis and Visualization of Multi-Scale Astrophysical Simulations Using Python and NumPy

Matthew Turk (mturk@slac.stanford.edu) – *KIPAC / SLAC / Stanford*, USA

The study the origins of cosmic structure requires large-scale computer simulations beginning with well-constrained, observationally-determined, initial conditions. We use Adaptive Mesh Refinement to conduct multi-resolution simulations spanning twelve orders of magnitude in spatial dimensions and over twenty orders of magnitude in density. These simulations must be analyzed and visualized in a manner that is fast, accurate, and reproducible. I present "yt," a cross-platform analysis toolkit written in Python. "yt" consists of a data-management layer for transporting and tracking simulation outputs, a plotting layer, a parallel analysis layer for handling mesh-based and particle-based data, as well as several interfaces. I demonstrate how the origins of cosmic structure – from the scale of clusters of galaxies down to the formation of individual stars – can be analyzed and visualized using a NumPy-based toolkit. Additionally, I discuss efforts to port this analysis code to other adaptive mesh refinement data formats, enabling direct comparison of data between research groups using different methods to simulate the same objects.

Analysis of Adaptive Mesh Refinement Data

I am a graduate student in astrophysics, studying the formation of primordial stars. These stars form from the collapse of large gas clouds, collapsing to higher densities in the core of extended star-forming regions. Astrophysical systems are inherently multi-scale, and the formation of primordial stars is the best example. Beginning with cosmological-scale perturbations in the background density of the universe, one must follow the evolution of gas parcels down to the mass scale of the moon to have any hope of resolving the inner structure and thus constrain the mass scale of these stars.

In order to do this, I utilize a code designed to insert higher-resolution elements within a fixed mesh, via a technique called adaptive mesh refinement (AMR). Enzo [ENZ] is a freely-available, open source AMR code originally written by Greg Bryan and now developed through the Laboratory for Computational Astrophysics by a multi-institution team of developers. Enzo is a patch-based multi-physics AMR/N-body hybrid code with support for radiative cooling, multi-species chemistry, radiation transport, and magnetohydrodynamics. Enzo has been used to simulate a wide range of astrophysical phenomena, such as primordial star formation, galaxy clusters, galaxy formation, galactic star formation, black hole accretion and jets from gamma ray bursts. Enzo is able to insert up to 42 levels of refinement (by factors of two) allowing for a dynamic range between cells of up to 2^{42}. On the typical length scale of primordial star formation, this allows us to resolve gas parcels on the order of a hundred miles, thus ensuring the simulations fully resolve at all times the important hydrodynamics of the collapse.

A fundamental but missing aspect of our analysis pipeline was an integrated tool that was transparently parallelizable, easily extensible, freely distributable, and built on open source components, allowing for full inspection of the entire pipeline. My research advisor, Prof. Tom Abel of Stanford University, suggested I undertake the project of writing such a tool and approach it from the standpoint of attacking the problem of extremely deep hierarchies of grid patches.

Initially, `yt` was written to be a simple interface between AMR data and the plotting package "HippoDraw," which was written by Paul Kunz at the Stanford Linear Accelerator Center [HIP]. As time passed, however, it moved more toward a different mode of interaction, and it grew into a more fully-featured package, with limited data management, more abstract objects, and a full GUI and display layer built on wxPython [WX] and Matplotlib [MPL], respectively. Utilizing commodity Python-based packages, I present a fully-featured, adaptable and versatile means of analyzing large-scale astrophysical data. It is based primarily on the library NumPy [NPY], it is mostly written in Python, and it uses Matplotlib, and optionally PyTables and wxPython for various sub-tasks. Additionally, several pieces of core functionality have been moved out to C for fast numerical computation, and a TVTK-based [TVTK] visualization component is being developed.

Development Philosophy

From its beginning, `yt` has been exclusively free and open source software, and I have made the decision that it will never require components that are not open source and freely available. This enables it to be distributed, not be dependent on licensing servers, and to make available to the broader community the work put forth by me, the other developers, and the broader contributing community toward approachable analysis of the data. The development has been driven, and will continue to be driven, by my needs, and the needs of other developers.

Furthermore, no feature that I, or any other member of the now-budding development team, implement will be hidden from the community at large. This philosophy has served the toolkit well already; it has already

Presented at the 7th Annual Python in Science Conference, 8/19/2008 to 8/24/2008, Pasadena, CA, USA
Work supported in part by US Department of Energy contract DE-AC02-76SF00515

been examined and minor bugs have been found and corrected.

In addition to these commitments, I also sought the ability to produce publication-quality plots and to reduce the difficulty of multi-step operations. The user should be presented with a consistent, high-level, interface to the data, which will have the side-effect of enabling different entry points to the toolkit as a whole.

The development of `yt` takes place in a publicly accessible subversion repository with a Trac frontend [YT]. Sphinx-based documentation is available, and automatically updated from the subversion repository as it is checked in. In order to ease the process of installation, a script is included to install the entire set of dependencies along with the toolkit; furthermore, installations of the toolkit are maintained at several different supercomputing centers, and a binary version for Mac OS X is provided.

Organization

To provide maximum flexibility, as well as a conceptual separation of the different components and tasks to which components can be directed, `yt` is packaged into several sub-packages.

The analysis layer, `lagos`, provides several features beyond mere data access, including extensive analytical capabilities. At its simplest level, `lagos` is used to access the parameters and data in a given time-based output from an AMR simulation. However, on top of that, different means of addressing collections of data are provided, including from an intuitive object-oriented perspective, where objects are described by physical shapes and orientations.

The plotting layer, `raven`, has capabilities for plotting one-, two- and three-dimensional histograms of quantities, allowing for weighting and binning of those results. A set of pixelization routines have been written in C to provide a means of taking a set of variable-size pixels and constructing a uniform grid of values, suitable for fast plotting in Matplotlib - including cases where the plane is not axially perpendicular, allowing for oblique slices to be plotted and displayed with publication-quality rendering. Callbacks are available for overlaying analytic solutions, grid-patch boundaries, vectors, contours and arbitrary annotation.

Additionally, several other sub-packages exist that extend the functionality in various different ways. The `deliverator` package is a Turbogears-based [TG] image gallery that listens for SOAP-encoded information about images on the web, `fido` stores and retrieves data outputs, and `reason` is the wxPython-based [WX] GUI.

Object Design and Protocol

One of the difficulties in dealing with rectilinear adaptive mesh refinement data is the fundamental disconnect between the geometries of the grid structure and the objects described by the simulation. One does not expect galaxies to form and be shaped as rectangular prisms; as such, access to physically-meaningful structures must be provided. To that end, `yt` provides the following:

- Sphere
- Rectangular prism
- Cylinder / disk
- “Extracted” regions based on logical operations
- Topologically-connected sets of cells

Each of these regional descriptors is presented to the user as a single object, and when accessed the data is returned at the finest resolution available; all overlapping coarse grid cells are removed transparently. This was first implemented as physical structures resembling spheres were to be analyzed, followed by disk-like structures, each of which needed to be characterized and studied as a whole. By making available these intuitive and geometrically meaningful data selections, the underlying physical structures that they trace become more accessible to analysis and study.

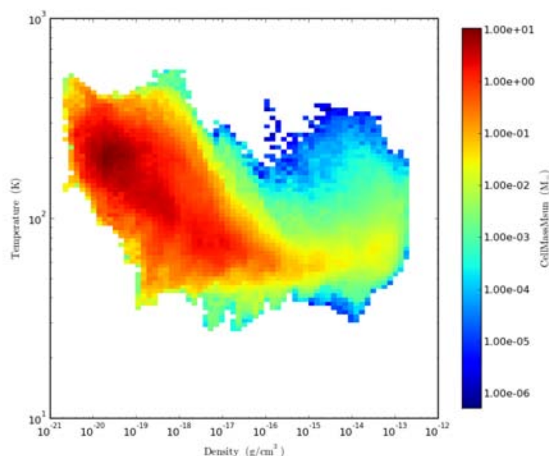
The objects are designed so that code snippets such as the following are possible:

```
>>> sp = amr_hierarchy.sphere(
...           center, radius)
>>> print sp["Density"].min()
>>> L_vec = sp.quantities["AngularMomentumVector"]()
>>> my_disk = amr_hierarchy.disk(center,
...           L_vec, radius, radius/100.0)
>>> print my_disk["Density"].min()
```

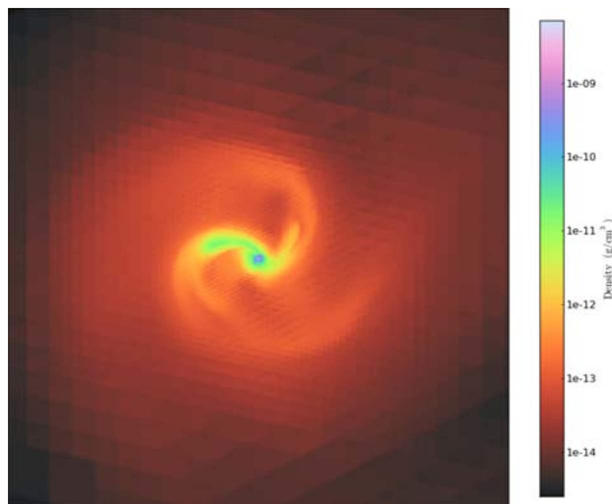
The abstraction layer is such that there are several means of interacting with these three-dimensional objects, each of which is conceptually unified, and which respects a given set of data protocols. Due to the flexibility of Python, as well as the versatility of NumPy, this functionality has been easily exposed in the form of multiple returned arrays of data, which are fast and easily manipulated. Above can be seen the calculation of the angular momentum vector of a sphere, and then the usage of that vector to construct a disk with a height relative to the radius.

These objects handle cell-based data fields natively, but are also able to appropriately select and return particles contained within them. This has facilitated the inclusion of an off-the-shelf halo finder, which allows users to quantify the clustering of particles within a region.

In addition to the object model, a flexible interface to derived data fields has been implemented. All fields, including derived fields, are allowed to be defined by either a component of a data file, or a function that transforms one or more other fields, thus allowing multiple layers of definition to exist, and allowing the user to extend the existing field set as needed. Furthermore, these fields can rely on the cells from neighboring grid patches - which will be generated automatically by `yt` as needed - which enables the creation of fields that rely on finite-difference stencils.



A two-dimensional phase diagram of the distribution of mass in the Density-Temperature plane for a collapsing gas cloud



An oblique slice through the center of a star formation simulation. The image plane is normal to the angular momentum vector.

The combination of derived fields, physically-meaningful data objects and a unique data-access protocol enables `yt` to construct essentially arbitrary representations of arbitrary collections of data. For instance, the user is able to take arbitrary profiles of data objects (radial profiles, probability distribution functions, etc) in one, two and three dimensions. These can be plotted from within the primary interface, and then output in a publication-ready format.

Two-Dimensional Data Representations

In order to make images and plots, `yt` has several different classes of two-dimensional data representations, all of which can be turned into images. Each of these objects generates a list of variable-resolution points, which are then passed into a C-based pixelization routine that transforms them into a fixed-resolution buffer, defined by a width, a height, and physical boundaries of the source data.

The simplest means of examining data is through the usage of axially-parallel slices through the dataset. This has several benefits - it is easy to calculate which grids and which cells are required to be read off disk (and most data formats allow for easy "striding" of data off disk, which reduces this operation's IO overhead) and it is easy to automate the process to step through a given dataset.

However, at some length scales in star formation problems, gas is likely to collapse into a disk, which is often not aligned with the axes of the simulation. By slicing along the axes, patterns such as spiral density waves could be missed, and ultimately go unexamined. In order to better visualize off-axis phenomena, I implemented a means of creating an image that is misaligned with the axes.

This "cutting plane" is an arbitrarily-aligned plane that transforms the intersected points into a new coordinate system such that they can be pixelized and made into a publication-quality plot. This technique required a new pixelization routine, in order to ensure that the correct voxels were taken and placed on the plot, which required an additional set of checks to determine if the voxel intersected with the image plane.

The nature of adaptive mesh refinement is such that one often wishes to examine either the sum of values along a given sight-line or a weighted-average along a given sight-line. `yt` provides an algorithm for generating line integrals in an adaptive fashion, such that every returned (x, y, v, dx, dy) point does not contain data from any points where $dx < dx_p$ or $dy < dy_p$.

We do this in a multi-step process, operating on each level of refinement in turn. Overlap between grids is calculated, such that, along the axis of projection, each grid is associated with a list of grids that it overlaps with on at least one cell. We then iterate over each level of refinement, starting with the coarsest, constructing lists of both "further-refinable" cells and "fully-refined" cells. A combination step is conducted, to combine overlapping cells from different grids; all "further-refinable" cells are passed to the next level as input to the overlap algorithm, and we continue recursing down the level hierarchy. The final projection object, with its variable-resolution cells, is returned to the user.

Once this process is completed, the projection object respects the same data protocol, and can be plotted in the same way, as an ordinary slice.

Contour Finding

Ofttimes, one needs to identify collapsing objects by finding topologically-connected sets of cells. The nature of adaptive mesh refinement, where in a given set cells may be connected across grid and refinement boundaries, requires sophisticated means for such identification.

Unfortunately, while locating topologically-connected sets inside a single-resolution grid is a straightforward but non-trivial problem in recursive programming, extending this in an efficient way to hierarchical datasets can be problematic. To that end, the algorithm implemented in yt checks on a grid-by-grid basis, retrieving an additional set of cells at the grid boundary. Any contour that crosses into these 'ghost zones' mandates a reconsideration of all grids that intersect with the currently considered grid. This process is expensive, as it operates recursively, but ensures that all contours are automatically joined.

Once contours are identified, they are split into individual derived objects that are returned to the user. This presents an integrated interface for generating and analyzing topologically-connected sets of related cells. In the past, yt has been used to conduct this form of analysis and to study fragmentation of collapsing gas clouds, specifically to examine the gravitational boundedness of these clouds and the scales at which fragmentation occurs.

Parallel Analysis

As the capabilities of supercomputers grow, the size of datasets grows as well. In order to meet these changing needs, I have been undertaking an effort to parallelize yt to run on multiple independent processing units. Specifically, I have been utilizing the Message Passing Interface (MPI) via the MPI4Py [MPI] module, a lightweight, NumPy-native wrapper that enables natural access to the C-based routines for interprocess communication. My goal has been to preserve at all times the API, such that the user can submit an unchanged serial script to a batch processing queue, and the toolkit will recognize it is being run in parallel and distribute tasks appropriately.

The tasks in yt that require parallel analysis can be divided into two different broad categories: those tasks that can act on data in an unordered, uncorrelated fashion, and those tasks that act on a decomposed image plane.

To parallelize the unordered analysis, a set of iterators have been implemented utilizing an initialize/finalize structure. Upon initialization of the iterator, it calls a method that determines which sets of data will be processed by which processors in the MPI group. The iteration proceeds as normal, and then, before the StopIteration exception is raised, it finalizes by broadcasting the final result to every processor. The unordered nature of the analysis allows the grids to be ordered such that disk access is minimized; on high-performance file systems, this results in close-to-ideal scaling of the analysis step.

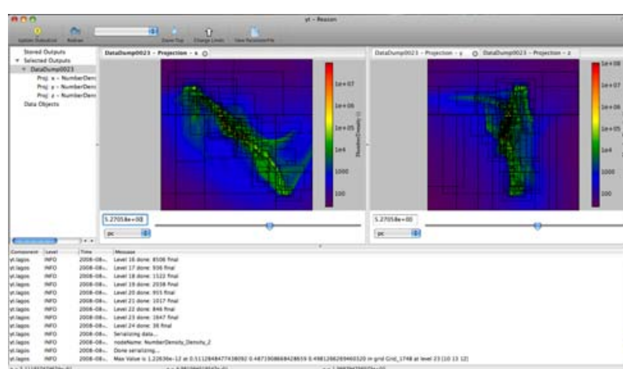
Constraints of Scale

In order to manage simulations consisting of multiple hundreds of thousands of discrete grid patches - as well as their attendant grid cell values - I have undertaken optimization using the `cProfile` module to locate and eliminate as many bottlenecks as possible. To that end, I am currently in the process of reworking the object instantiation to rely on the Python feature 'slots,' which should speed the process of generating hundreds of thousands of objects. Additionally, the practice of storing data about simulation outputs between instantiation of the Python objects has been extended; this speeds subsequent startups, and enables a more rapid response time.

Enzo data is written in one of three ways, the most efficient -and prevalent- way being via the Hierarchical Data Format (HDF5) [HDF] with a single file per processor that the simulation was run on. To limit the effect that disk access has on the process of loading data, hand-written wrappers to the HDF5 have been inserted into the code. These wrappers are lightweight, and operate on a single file at a time, loading data in the order it has been written to the disk. The package PyTables was used for some time, but the instantiation of the object hierarchy was found to be too much overhead for the brief and well-directed access desired.

Frontends and Interfaces

yt was originally intended to be used from the command line, and images to be viewed either in a web browser or via an X11 connection that forwarded the output of an image viewer. However, a happy side-effect of this attitude - as well as the extraordinarily versatile Matplotlib "Canvas" interface - is that the yt API, designed to have an a single interface to analysis tasks, is easily accessed and utilized by different interfaces. By ensuring that this API is stable and flexible, GUIs, web-interfaces, and command-line scripts can be constructed to perform common tasks.



A typical session inside the GUI

For scientific computing as a whole, such flexibility is invaluable. Not all environments have access to the same level of interactivity; for large-scale datasets, being able to interact with the data through a scripting interface enables submission to a batch processing queue, which enables appropriate allocation of resources. For smaller datasets, the process of interactively exploring datasets via graphical user interfaces,

exposing analytical techniques not available to an offline interface, is extremely worthwhile, as it can be highly immersive.

The canonical graphical user interface is written in wxPython, and presents to the user a hierarchical listing of data objects: static outputs from the simulation, as well as spatially-oriented objects derived from those outputs. The tabbed display pane shows visual representations of these objects in the form of embedded Matplotlib figures.

Recently an interface to the Matplotlib 'pylab' interface has been prepared, which enables the user to interactively generate plots that are thematically linked, and thus display an uniform spatial extent. Further enhancements to this IPython interface, via the profile system, have been targeted for the next release.

Knoboo [KBO] has been identified as a potential web-based interface, in the same style as Sage. It is a lightweight software package designed to display executed Python code in the browser but to conduct the execution on the backend. With a disjoint web-server and execution kernel model, it enables the frontend to communicate with a remote kernel server where the data and analysis packages would reside. Because of its flexibility in execution model, I have already been able to conduct analysis remotely using Knoboo as my user interface. I intend to continue working with the Knoboo developers to enhance compatibility between `yt` and Knoboo, as web-based interfaces are a powerful way to publish analysis as well as to enable collaboration on analysis tasks.

Generalization

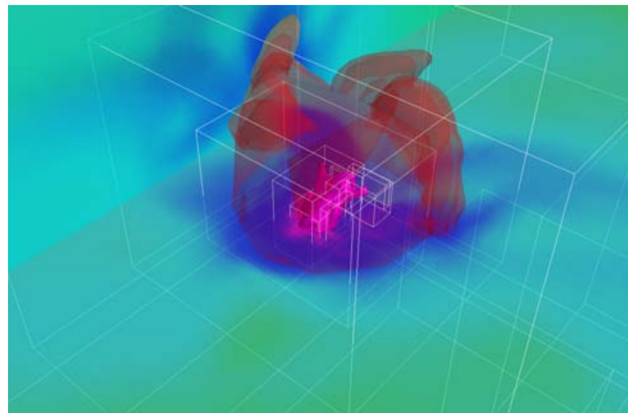
As mentioned above, `yt` was designed to handle and analyze data output from the AMR code Enzo. Dr. Jeff Oishi of Berkeley is leading the process of converting the toolkit to work equally well with data from other AMR codes; however, different codes make separate sets of assumptions about outputted data, and this must be generalized to be non-Enzo specific.

In this process, we are having to balance a desire for generalization with a desire for both simplicity and speed. To that extent, we are attempting to make minimally invasive changes where possible, and rethinking aspects of the code that were not created in the most general fashion.

By providing a unified interface to multiple, often competing, AMR codes, we will be able to utilize similar - if not identical - analysis scripts and algorithms, which will enable direct comparison of results between groups and across methods.

Future Directions

As the capabilities of `yt` expand, the ability to extend it to perform new tasks extend as well. Recently, the beginnings of a TVTK-based frontend were implemented, allowing for interactive, physically-oriented 3D visualization. This relies on the vtkCompositeDataPipeline object, which is currently weakly supported across the VTK codebase. However, the power of TVTK as an interface to VTK has significant promise, and it is a direction we are targeting.



A visualization within `yt`*, using the TVTK toolkit to create 3D isocontours and cutting planes.*

Work has begun on simulated observations from large-scale simulations. The first step toward this is simulating optically thin emissions, and then utilizing an analysis layer that operates on 2D image buffers.

By publishing `yt`, and generalizing it to work on multiple AMR codebases, I hope it will foster collaboration and community efforts toward understanding astrophysical problems and physical processes, while furthermore enabling reproducible research.

Acknowledgments

I'd like to acknowledge the guidance of, first and foremost, my PhD advisor Prof. Tom Abel, who inspired me to undertake this project in the first place. Additionally, I'd like to thank Jeff Oishi, a crucial member of the development team, and Britton Smith and Brian O'Shea, both of whom have been fierce advocates for the adoption of `yt` as a standard analysis toolkit. Much of this work was conducted at Stanford University and the Kavli Institute for Particle Astrophysics and Cosmology, and was supported (in part) by U.S. Department of Energy contract DE-AC02-76SF00515.

References

[ENZ] http://lca.ucsd.edu/projects/enzo
[HIP] http://www.slac.stanford.edu/grp/ek/hippodraw/
[WX] http://wxpython.org/
[MPL] http://matplotlib.sf.net/
[NPY] http://numpy.scipy.org/
[TVTK] http://svn.enthought.com/enthought/wiki/TVTK
[YT] http://yt.enzotools.org/
[TG] http://turbogears.org/
[MPI] http://mpi4py.scipy.org/
[HDF] http://hdfgroup.org/
[KBO] http://knoboo.com/