Introducing yt 4.0: Analysis and Visualization of Volumetric Data

This manuscript ([permalink](#)) was automatically generated from [yt-project/yt-4.0-paper@90d6f3c](#) on January 9, 2023.

Authors

- **The yt Project**
  · ⓘ yt-project · 🐦 yt_astro
  NumFOCUS

- **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

- **Nathan J Goldbaum**
  ⓘ 0000-0001-5557-267X · ⓖ ngoldbaum · 🐦 njgoldbaum
  National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign

- **John A. ZuHone**
  ⓘ 0000-0003-3175-2347 · ⓖ jzuhone · 🐦 astrojaz
  Harvard-Smithsonian Center for Astrophysics

- **Cameron Hummels**
  ⓘ 0000-0002-3817-8133 · ⓖ chummels
  Department of Theoretical Astrophysics, California Institute of Technology

- **Suoqing Ji**
  ⓘ 0000-0001-9658-0588 · ⓖ jisuoqing
  Physics Department, University of California Santa Barbara

- **Meagan Lang**
  ⓘ 0000-0002-2058-2816 · ⓖ langmm
  National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign

- **Madicken Munk**
  ⓘ 0000-0003-0117-5366 · ⓖ munkm
  National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign

- **Britton Smith**
  ⓘ 0000-0002-6804-630X · ⓖ brittonsmith
  University of Edinburgh

- **Kacper Kowalik**
  ⓘ 0000-0003-1709-3744 · ⓖ Xarthisius
  National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign

- **Miguel de Val-Borro**
  ⓘ 000-0002-0455-9384 · ⓖ migueldvb
  Planetary Science Institute

- **Jared W. Coughlin**
  iD [0000-0002-4373-4114](#) · [jcoughlin11](#)
  National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign

- **Corentin Cadiou**
  iD [0000-0003-2285-0332](#) · [cphyc](#) · [cphyc](#)
  Department of Physics and Astrophysics, University College London; Institut d'Astrophysique de Paris

- **Michael Zingale**
  iD [0000-0001-8401-030X](#) · [zingale](#)
  Stony Brook University

- **Leigh Orf**
  iD [0000-0002-2677-6427](#) · [leighorf](#)
  Space Science and Engineering Center, University of Wisconsin - Madison

- **Kelton Halbert**
  iD [0000-0001-6898-2731](#) · [keltonhalbert](#)
  Cooperative Institute for Meteorological Satellite Studies, The University of Wisconsin, Madison

- **Clément Robert**
  iD [0000-0001-8629-7068](#) · [neutrinoceros](#)
  Institut de Planétologie et d'Astrophysique de Grenoble (IPAG), Université Grenoble Alpes, France

- **Christopher Havlin**
  iD [0000-0003-0585-8236](#) · [chrishavlin](#)
  School of Information Sciences, University of Illinois at Urbana-Champaign

- **Stephanie Tonnesen**
  iD [0000-0002-8710-9206](#) · [stonnes](#)
  Center for Computational Astrophysics, Flatiron Institute

- **Andrew Myers**
  iD [0000-0001-8427-8330](#) · [atmyers](#)
  Lawrence Berkeley National Laboratory

- **Add Yourself**
  iD [XXXX-XXXX-XXXX-XXXX](#) · [yournamehere](#)
  Your University · Funded by Grant XXXXXXXX

Abstract

We present the current version of the `yt` software package. `yt` is an open-source, community-developed platform for analysis of volumetric data, with readers for several dozen data formats, indexing systems for gridded data, adaptive mesh refinement data, unstructured mesh data, discrete and particle formats, and octree-based data, as well as the combination of these. We describe the systems implemented in `yt` to facilitate a "science-first" approach to data analysis, wherein the emphasis is on the meaning and interpretation of the data as opposed to its discretization or layout.

Authorship Policy

We note that the author list for this paper is, by design, extensive. We have separated the authors into those that contributed to the text (whose names are ordered **somehow TBD**) and those that are members of the `yt` community. The authors from each group have been indicated in the respective author affiliations.

This paper was developed collaboratively, using the Manubot [1] system for collaborating on and reviewing contributed text.

> To add yourself to the author list, please follow the instructions in our README.

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 [2], 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 representatives from other domains are increasingly 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. Sections of the code base receiving such improvements include unit handling, plotting code, 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 participating 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 writing 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 among 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.

Project Governance

Between the publication of the first paper and this paper, the `yt` project instituted a form of governance involving a steering committee, a set of "members" of the project, and a defined process for

developing improvements and enhancements (the YTEP, or `yt`-enhancement-proposal process). YTEPs are discussed in 1.4.6. The systems developed account for a number of important procedures, mostly related to decision-making, but do not address pressing community needs such as community standards for conduct, changes in committee composition, sub-project coordination, or the transition of members and developers to "emeritus" status.

This governance structure, however, serves as a process for the `yt` *project*, rather than just the software itself. Much like other software projects with community-building as core goals, the `yt` project encompasses the core package itself (largely the subject of this paper), affiliated software projects (such as [3]) but also, crucially, the broader community and the spaces in which discussions, tutorials, inquiries and development take place. As such, the governance structure was designed to be more holistic than addressing so-called "Commit Privileges" and access control.

There are several components of the yt governance structure that we highlight here, but direct the interested reader to our governance source repository for a more detailed understanding.

Conflicts of Interest

`yt` project development is distributed across universities and organizations, where in some cases different members of the development team may find themselves collaborating, or even applying for similar funding or academic positions. To minimize real or perceived conflicts of interest among the community, a policy regarding conflicts of interest is applied to members of the `yt` steering committee.

> It is expected that the Steering Committee Members will be employed at a wide range of companies, universities and non-profit organizations. Because of this, it is possible that Members will have conflict of interests. Such conflict of interests include, but are not limited to:
>
> - Financial interests, such as investments, employment or contracting work, outside of The Project that may influence their work on The Project.
> - Access to proprietary information of their employer that could potentially leak into their work with the Project.
>
> All members of the Steering Committee shall disclose to the rest of the Council any conflict of interest they may have. Members with a conflict of interest in a particular issue may participate in Committee discussions on that issue, but must recuse themselves from voting on the issue.

This also addresses the possibility of funded work conflicting with the development of `yt` as a whole, or project wellness. As of the time of writing, this policy has not been exercised, but providing it in writing will ensure that the conduct of the steering committee is appropriate.

Project Members and Groups

Individuals interact with projects in different ways at different times; as careers progress and personal lives shift, the degree of engagement that one can expect will change as well. To reflect this, as well as to codify the different means by which people can participate in the community, the `yt` governance structure allows for "members" to transition between different levels of activity.

The base level of "membership" in the `yt` project is that of "project members." This is, by design, a broad category of individuals who have demonstrated their willingness to participate in the community, either through participating in community discussions, helping other members, contributing code or documentation, or reviewing the contributions of others. The initial seed of the project members was chosen by a flat cutoff of 50 changesets included in the repository; since that time, members are

nominated by other members and must receive at least three positive votes. This membership, unless either explicitly revoked (via five supporting member votes and a steering council vote) lasts for life, although may be transitioned into "emeritus" status. Within the "members" group, a set of several sub-groups are identified that follow a rough, orchard-like metaphor.

The first of these groups is the **fertilizer** group. (We whimsically note that the choice of name is not meant as a value judgment!) This is the high-level organizer group, including the steering committee and any other leadership roles. This group tends to the growth of the community, ensuring that there are opportunities for new members, and that the project continues as expected. Explicitly identified in the governance structure is that this is a group meant to train new members and to facilitate its membership to grow and change over time.

The **arboretum** is the largest group, and is the starting point for new members of the project. The arboretum includes all project members and steering committee members. This group facilitates and develops contributions to the code base, encouraging diversity of participation.

The **greenhouse** group includes those new members of the project, who are still developing their participation but have contributed to the community (for instance with a bug fix or documentation enhancement.) Members of the greenhouse group do not necessarily have high-level project responsibilities, but are encouraged to seek involvement and participate in triage meetings and other discussions.

In recognition that temporary obligations, changes in activity level and other interest migrations happen, there is a group entitled the **fallow fields** for members who wish to temporarily step back from their responsibilities. Many different reasons exist for stepping into the fallow fields – burnout, new or enhanced work responsibilities, family or care duties, or even simply a desire to do other things. Transitioning to the fallow fields is not permanent, and participation in the project can either resume or the individual can choose to move on.

Finally, we have designated a class of project members as **emeritus** (the only non-orchard name utilized!) for individuals who have moved on to other interests or activities. The yt project has had a number of individuals and contributors who have later gone on to work in industry or non-profits, or who simply have opted to leave the community. The emeritus designation is a manner by which their previous contributions can still be recognized and honored, while also recognizing that they are no longer active participants.

Decision Making

The majority of decisions in the yt project are small, day-to-day decisions about code quality, merging pull requests, how to respond to questions and other items for which it is typically quite easy to identify a consensus. For those situations where consensus is not possible, deciding upon and implementing a course of action can be one of the more difficult issues in a project; members of the community each bring their own preferences, requirements, needs and desires to a project. These decisions might be regarding the future of specific areas of the project's code base – for instance, "should the software volume rendering be split into a separate repository?" or "how should we decide the line to draw for excising old frontends?" Typically, the YTEP process is sufficient, even in cases where a consensus cannot be reached, as it allows a large amount of discussion. In cases where it cannot, the decision making process utilizes a multi-tier system of votes, including the YTEP and mailing list as venues. If a 2/3 majority is unable to be reached, the steering committee is allowed approximately a month to identify a consensus among themselves; failing this, a majority vote will be reached.

While the full mechanism of the yt project decision making process has not been exercised in its entirety, its presence has provided a "safety net" for discussion. For further details, the [voting process](#) section of the [governance documents](#) lay out the steps, involved groups and timeline.

Development Procedure

`yt` is developed openly. During the Spring of 2017, development transitioned from occurring on [Bitbucket](#) to [GitHub](#), and the source code management system was changed from [Mercurial](#) to [git](#). Development occurs through the "pull request" model, wherein changes to the codebase are made and then requested to be included in the primary repository. Typically, there are two branches of development, and occasionally three. The first of these is the "stable" branch, which is much slower-paced, and typically only modified during the release periods. The second is that of "main" (formerly "master", which is the conventional term in git terminology, and renamed in early 2021; the corresponding mercurial term would be "default") which is where current development takes place. The "main" branch is meant to be for development proceeding that does not drastically disrupt usage patterns. Occasionally, such as during the development of `yt` 4.0, a third branch is included in the primary repository. This development branch is open for large and potentially disruptive changes, but in order to centralize code review and developer attention it takes place there. For instance, during the development of `yt` 4.0, the branch `yt-4.0` was where the global mesh was removed and where the units subsystem was removed and replaced with `unyt`.

This three-pronged approach generally has suited the community; the process of backporting changes from the "main" branch to the "stable" branch can be time-consuming. However, balancing the needs of a community requiring stable methods for analyzing data against the ease of development suggests that this is a toll worth paying.

In general, the development of `yt` is reasonably top-heavy, with the majority of contributions coming from a core group of individuals. We discuss the implications of this on sustainability in Section 1.19, and provide here a graph of the contributions over time.
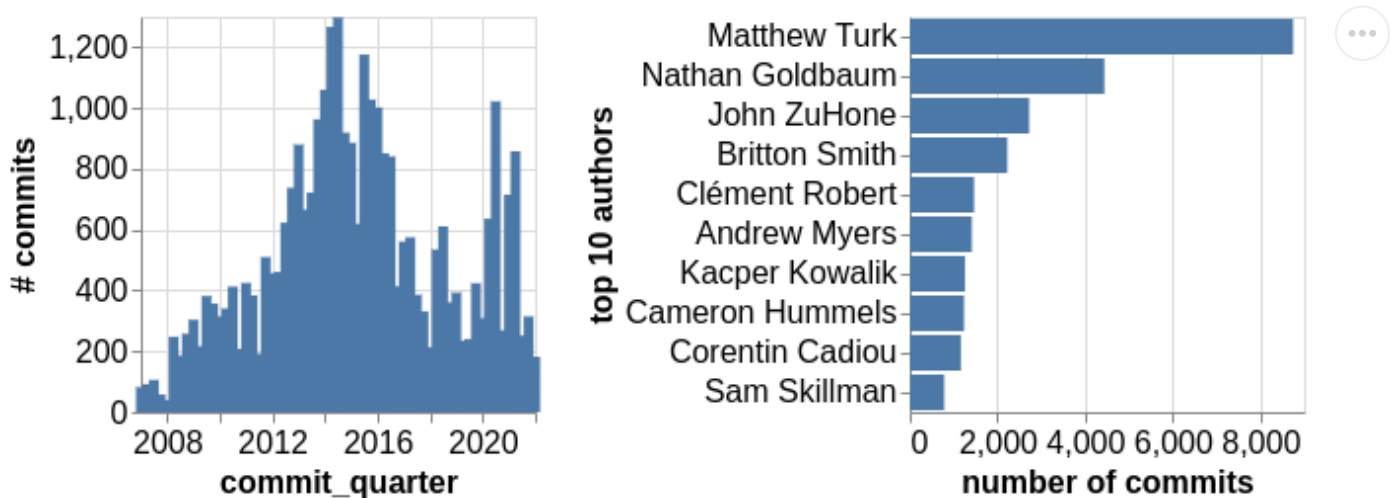

Figure 1: Commits to the source code as a function of time.

Unit Testing

The `yt` codebase includes a number of unit tests; although extensive, their existence post-dates the initial development of the code, and they largely work around the extant APIs at the time of their creation. Most modern recommendations for developing scientific software emphasize isolated components, well-structured interfaces, and few side effects. While the development process attempts to emphasize development of isolated APIs and well-constrained unit tests, the balance struck between enabling contribution from junior developers and ensuring the (subjective) standards of the code base does not always fall on the side of rigid design.

Many of the `yt` APIs that are tested require the existence of a "dataset." For instance, the testing of whether objects are correctly selected by a sphere selector (which absolutely *could* be tested in isolation, were the APIs more separable) is done via creating several different sets of mock datasets of different organizations and shapes and testing whether or not they correctly choose the data points to be included. To support these operations, the `yt` testing utilities provide helper functions for creating mock datasets that have different geometric configurations and different collections of "fields" included in their set of primitive values. Many of the tests are parameterized against the types and organizations of the datasets, the decomposition across mock processors, and the underlying values of the fields. This ensures that we check against errors and bugs that may depend on behavior that varies as the number of processors or the organization of the data changes. One example of this would be in the selection of grid values for a single grid of size 128^3. The values selected in this should match the values selected in the same grid decomposed into eight sets of 64^3 cells, or 64 sets of 32^3 cells.

The mechanism by which fields are tested is somewhat more extensive, touching on two different needs. The first need is that of accuracy – fields with known answers, or fields that can be written to be decomposed into primitive, non-optimized operations, are tested for correctness. The second need is that of dependency calculation; all fields should have their dependencies correctly detected. For example, if a dataset has primitive fields for "mass" and "velocity," the calculation of momentum should require both. If the dataset includes a "momentum" field, then that should be detected as well. This dependency calculation enables `yt` to consolidate IO tasks and read as much data as possible in each pass over the full dataset. In addition to this, fields are tested to ensure that the values generated for them are independent of the organization of the dataset. Like in the example above, the "momentum" field for a fixed set of values should be identical regardless of the decomposition of the individual cell elements.

Wherever possible, analytical solutions are preferred. For processes like surface extraction, this might include ensuring that fixed radii extraction produce the correct spherical region. For streamlines, it might include computing the analytical solution to an integration along a known vector field. And for projections, it would mean that integrating the path with a weight of "one" should result in a uniform set of values equal to the path length across the domain.

At present, the unit tests in `yt` take a considerable amount of time to run, and are using the nosetests framework. Modern Python practice is to use the newer pytest framework, and efforts are underway to port `yt` to utilize pytest, and in the process, attempt to reduce overall runtime.

Answer Testing

The most time-consuming part of the testing process is what we refer to as "answer testing." Because so much of `yt` is focused on computing analysis results, and because some of these analysis results simultaneously depend on specific IO routines, selection routines, and many "frontend-specific" pieces of code, we have built a system for ensuring that for a given set of analysis operations, the result of a set of operations does not change beyond a fixed (typically quite small) tolerance.

In general, we allow three different classes of answers, against which we compare results from the current version of the code:

1. Data values which should not ever change unless an explicit decision is made (i.e., raw data values accessed from on-disk fields)
2. Lightly-processed data values which we do not anticipate any deviation from exact, ordered values (i.e., averages, extrema, etc.)
3. Higher-level processed values which may differently accumulate error across platforms and architectures but are still within fine-grained ($\equiv 10^{-7}$) tolerance (i.e., images, pixelized projections, etc.)

In the first case, we can utilize hashing functions (such as MD5 and SHA) to guarantee consistency across executions. Typically, however, we store the full set of values to facilitate easy comparison. In the latter two cases, we apply unit-aware relative tolerances. This allows for changes in unit assignment to be isolated from changes in value, and furthermore allows minor migration of values. A recent incident in which these tests needed to be changed resulted from minor differences as a result of consolidating operations within a loop to conserve memory; the code in question was converted to Cython and the drift was on the scale of 10^{-10}.

For small-scale answer tests, results are stored in a separate repository that is cloned as a subrepository of the principle `yt` repository. When a new set of answers are needed, they are submitted via pull request, and the changeset hash used for answer validation is updated in the main repository. This allows a different cadence, and also enables individuals *not* interested in updating answer values to avoid cloning the subrepository and its full history. Larger dataset answers are stored in on our continuous integration servers; a YAML file in the main `yt` repository stores the current version number for those answers, which is incremented (via a pull request) when needed. Requiring this clear decision-making process allows for both collaborative discussion and community governance over the degree of answer drift allowed.

Code Review

Code review in `yt` is conducted on a line-by-line basis, as well as on a higher-level regarding pull requests. The workflow for code review roughly follows this outline:

1. A pull request is issued. When a new pull request is issued, a template is provided that includes a description of the change, requesting information about its compliance with coding standards, etc.
2. The pull request is automatically marked as unmergeable until a team member applies the correct component label.
3. Code is reviewed, line-by-line, and suggestions are made by humans. Code linting is automated.
4. This process is iterated, ensuring that tests, results accuracy and coding standards are maintained.

One increasing issue with the code review process is ensuring that changes are reviewed with appropriate urgency; larger pull requests tend to languish without review, as the requirements for review necessarily add burden to the maintainers. "Bugfix" changes formally require only one reviewer, whereas the `yt` guidelines suggest that larger changes require review from two different team members.

One of the most pressing bottlenecks with code review is that the time it takes for tests to pass is much longer than the typical time span during which code review takes place. Because tests are often required to be run on the *current* version of the code, not the version of the code against which the pull request has been issued, they are often re-initiated following a merge. This results in a pull request being merged, and then whatever pull request is next to be reviewed has to wait until the tests (now updated with the newly accepted pull request) pass. To alleviate this, we have recently begun utilizing the "auto-merge" feature provided by GitHub. This allows a maintainer to mark a pull request as "queued" to be merged once a set of requirements – such as tests passing, approval messages, comment resolution and so forth – are met. By queuing up pull requests for merging, it allows maintainers to mark a set of pull requests as ready to be merged, and then when they meet the appropriate (automated and asynchronous) criteria, they will be merged.

Code Styling and Linting

For code included in `yt`, a set of styles are enforced.

We rely on the `pre-commit` framework, which enables automated checks as well as automatic fixes at commit time. This tool is an opt-in so not every "drive-by" contributor has to learn and install it, but continuous linting is provided by pre-commit.ci so styling errors cannot slip in.

We configure `pre-commit` to run a mixture of formatters and static checkers. The former modify the code in place, while the latter only report errors and so-called "code smells" (such as undefined variables, unused imports, bare `except` statements…) but require human developers fix them.

Our suite of formatter most proeminently includes `black`, `isort` and `pyupgrade`. `black` has been designed to maximize readability with as few free parameters as possible (In many ways, the fact that most of the `yt` code developers did not utilize this style before it was enforced likely enabled its uptake, as it was seen as a choice that "made everyone compromise."), while `isort` ensures that all `import` statements are sorted (according to alphabetical order within a first/second/third-party categorization), and `pyupgrade` modernizes some Python idioms according to our minimal support version of the language. In particular, `pyupgrade` enforces the usage of modern Python "f-strings" since we do not support Python version older than 3.6.

For static code analysis we rely on the `flake8` framework. Effort is underway to enable using `mypy` for the specialized task of type checking.

All changes that can be applied via automation (specifically, code formatting) are accessible from within the GitHub pull request interface, and are again provided by pre-commit.ci. This allows drive-by contributions to have their pull requests updated inline by an automated process, reducing the need to manually install packages to apply the changes.

Type Hinting

Code included in `yt` is not required to utilize type hinting. However, new contributions are allowed to include type hinting, and work is underway to develop an ontology of types as well as a base set of utilities and types that would be used both internally, and possibly be made available to external package developers.

YTEP Process

YTEPs, or "`yt`-enhancement proposal" are vehicles for collaborative decision-making in the project. Implemented shortly after the first paper on `yt` was released, the YTEP process experienced a fairly pronounced period of usage during the transition between versions 2.0 and 3.0 of `yt`, and has since been utilized considerably less. During periods of rapid development, the needs of the community for stability have to be balanced against desires for change; the YTEP process was implemented to facilitate stakeholder feedback, allow for discussion of design decisions, and to prompt detailed thinking about how and why things should be implemented. We have modeled this process against that used in the AstroPy community ("APE"). To create a new proposal for a large change to `yt`, or to document a decision-making process, individuals prepare a description of the background, motivation for the change, the steps to implementation, and potential alternative approaches. The proposal is discussed through the pull-request process, and once discussion has concluded it is added to the repository of YTEPs that is auto-built and deployed. The accepted YTEPs have included implementing the chunking system, developing a units system, removing legacy components, and implementing a code of conduct.

Below, we include a table of current YTEPs as of this writing.

Number	YTEP Title	Created	Authors

Number	YTEP Title	Created	Authors
0001	IO Chunking	November 26, 2012	Matthew Turk
0002	Profile Plotter	December 5, 2012	Matthew Turk
0003	Standardizing field names	December 11, 2012	Casey Stark, Nathan Goldbaum, Matthew Turk
0005	Octrees for Fluids and Particles	December 24, 2012	Matthew Turk
0006	Periodicity	January 10, 2013	Matthew Turk, Nathan Goldbaum
0007	Automatic Pull Requests' validation	February 21, 2013	Kacper Kowalik
0008	Release Schedule	February 21, 2013	Matthew Turk
0009	AMRKDTree for Data Sources	February 28, 2012	Sam Skillman
0010	Refactoring for Volume Rendering and Movie Generation	March 3, 2013	Cameron Hummels
0011	Symbol units in yt	March 7, 2013	Nathan Goldbaum, Casey Stark, Anna Rosen, Matthew Turk
0012	Halo Redesign	March 7, 2013	Britton Smith, Cameron Hummels, Chris Moody, Mark Richardson, Yu Lu
0013	Deposited Particle Fields	April 25, 2013	Chris Moody, Matthew Turk, Britton Smith, Doug Rudd, Sam Leitner
0014	Field Filters	July 2nd, 2013	Matthew Turk
0015	Transfer Function Refactor	August 13, 2013	Sam Skillman
0016	Volume Traversal	September 10, 2013	Matthew Turk
0017	Domain-Specific Output Types	September 18, 2013	Matthew Turk and Anthony Scopatz
0018	Changing dict-like access to Static Output	September 18, 2013	Matthew Turk
0019	Reduce items in main import	October 2, 2013	Matthew Turk
0020	Removing PlotCollection	March 18, 2014	Matthew Turk
0021	Particle-Only Plots	August 29, 2014	Andrew Myers
0022	Benchmarks	January 19, 2015	Matthew Turk
0023	yt Community Code of Conduct	July 11, 2015	Britton Smith
0024	Alternative Smoothing Kernels	August 1, 2015	Bili Dong

Number	YTEP Title	Created	Authors
0025	The ytdata Frontend	August 31, 2015	Britton Smith
0026	NumPy-like Operations	September 21, 2015	Matthew Turk
0027	Non-Spatial Data	December 1, 2015	Matthew Turk, Nathan Goldbaum, John ZuHone
0028	Alternative Unit Systems	December 8, 2015	John ZuHone, Nathan Goldbaum, Matthew Turk
0029	Extension Packages	January 25, 2016	Matthew Turk
0031	Unstructured Mesh	December 18, 2014	Matthew Turk
0032	Removing the global octree mesh for particle data	February 9 2017	Nathan Goldbaum, Meagan Lang, Matthew Turk
0033	Dropping Python2 Support	November 28, 2017	Nathan Goldbaum
0034	yt FITS Image Standard	September 9, 2018	John ZuHone
0036	Migrating from nose to pytest	September 30, 2019	Jared Coughlin
0037	Code Styling	May 18, 2020	Clément Robert
1000	GitHub Migration	March 25, 2017	Lots of folks
1776	Team Infrastructure	August 24, 2014	Britton Smith
3000	Let's all start using yt 3.0!	October 30, 2013	Matthew Turk

Indexing and Geometry

yt is designed for analysis and visualization of datasets that describe "natural" or "physical" phenomena; more generally, yt is designed to analyze data that can be characterized by a metric of some type. The most common use case, by far, is that of data that is described in a Cartesian space, by the orthogonal axes of x, y and z. However, for reasons related to naturalness of coordinate systems and relevance to physical phenomena, datasets are also frequently organized in other coordinate systems, such as cylindrical polar (r, z and θ), spherical (r, θ and ϕ) and variants such as geographic (latitude, longitude and altitude).

Importantly, however, yt distinguishes between the *coordinate* space a dataset describes and the natural or *index* space by which its organization is described. This distinction is the most relevant among datasets and data formats where the organization is *implicit*, rather than *explicit*; for instance, in a grid patch dataset, data variable locations are often only specified implicitly. For a grid volume that covers a given region, the relationship between the "index" value of a cell (for instance, i, j, k) and its position in space (for instance, x, y, z or r, θ, ϕ) requires transformation between a logically-Cartesian decomposition of the space and the potentially-non Cartesian space that it represents.

In Figure [2](#) we demonstrate one possible mapping. We note that the specific data layout is not optimized for IO throughput, and is unlikely to be exactly replicated in real world formats. In this case, the data points may be laid out sequentially on disk (or in memory) and a mapping function translates these into position and extent in the coordinate system, here cylindrical coordinates. For instance, there may be a cell that spans r from `0.375 to 0.5` and θ from `45.0 to 52.5`, which is defined by the array values defined in cell `1, 4`.
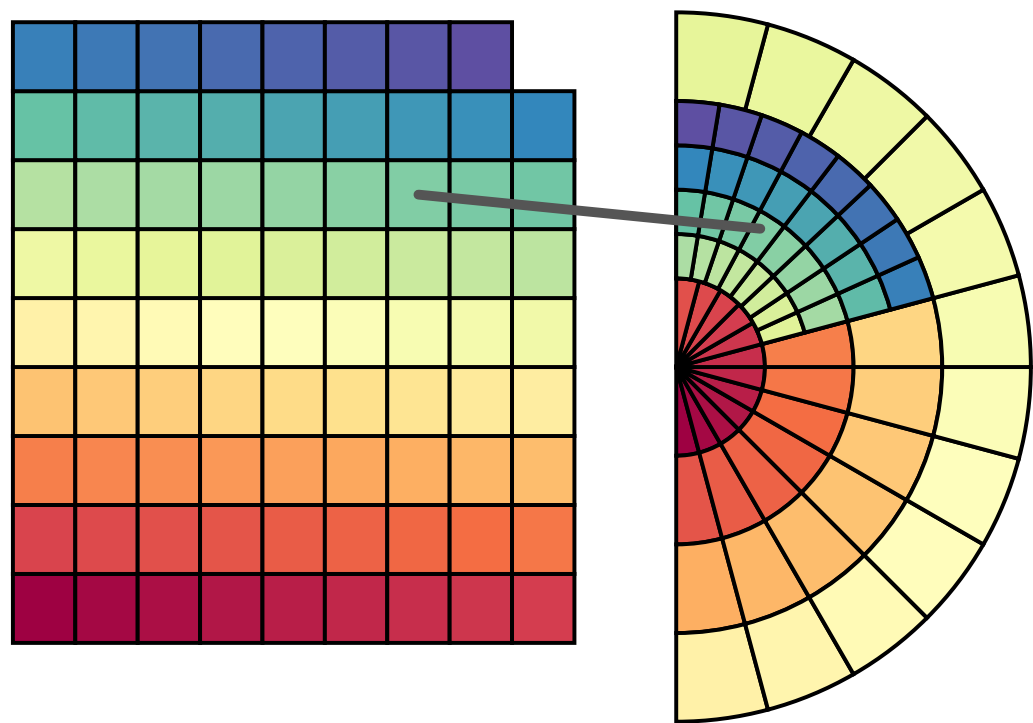
Figure 2: Index space to coordinate space mapping. On the left is an example of how data points may be laid out on disk and on the right is how these points might be translated into a (cylindrical) coordinate space.

Abstraction of Coordinate Systems

yt provides a system for defining relationships between index-space and coordinate-space. During instantiation of a `Dataset` object, a helper object (`coordinates`, a subclass of `CoordinateHandler`) is created. This helper object tracks the correspondence between numerical axes and spatial axes (for instance, even in some Cartesian datasets, axis 0 corresponds to z rather than x), the names of axes, and the transformation and pixelization methods for visualization. In addition to these helper functions, the coordinate handler provides definitions for derived fields that describe local cell width (and orthogonal path length), positions in coordinate space as computed by index space coordinates, volumes, and surface areas. These coordinate handlers also provide transformations between different spaces, albeit using the somewhat undesirable method of conversion to reference cartesian frames and subsequent conversion to local coordinate frames.

At present, coordinate spaces are defined in the spaces enumerate in Table [1](#). While these are representative of the most common spatial representations, additional representations (such as those that include a non-trivial mapping between coordinates and index values) are possible to implement.

Table 1: Extant coordinate systems; in all cases, value ranges should be taken to describe extent rather than specific boundary points.

Coordinate system	Axes
Cartesian coordinates	x, y, z

Coordinate system	Axes
Cylindrical polar coordinates	$r, \theta \in [0, 2\pi], z$
Spherical coordinates	r, θ, ϕ
Geographic coordinates	latitude $\in [0, 180]$, longitude $\in [0, 360]$, altitude
Internal geographic coordinates	latitude, longitude, depth
Spectral cube	Image x, Image y and ν

Future developments may involve code generation for arbitrary coordinate systems, using SymPy or other libraries. Independent of the visualization methods (which can often be reused), the development of coordinate systems is largely rote, applying straightforward mathematics to construct derived field definitions. As such, using mechanisms in SymPy for construction of relationships between coordinate systems may be a feasible method of developing code-generation for coordinate system handlers in yt.

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.

Selections in `yt` are usually 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.

Selection Routines

Given these set of hierarchical selection methods, all of which are designed to provide opportunities for early-termination, each *geometric* selector object is required to implement a small set of methods to expose its functionality to the hierarchical selection process. Duplicative functions often result from attempts to avoid expensive calculations that take into account boundary conditions such as periodicity and reflectivity unless necessary. Additionally, by providing some routines as options, we can in some instances specialize them for the specific geometric operation.

- `select_cell(cell_center, cell_width)` : this function, which is somewhat degenerate with `select_bbox` , returns whether a given "cell," defined by its center and its width along each dimension, is included within the selection. In situations where the cells are spaced logarithmically, rather than linearly, this may produce slightly reduced accuracy for near-misses and glancing-selections.
- `select_point(position)` : this function returns whether or not a point of zero-extent is included within the selection. This has some degeneracy with `select_sphere` .
- `select_sphere(position, radius)` : This is equivalent to the `select_point` function, except that any point within the specified radius is included within the selector object.
- `select_bbox(lower_left, upper_right)` : Determine overlap with an axis-aligned bounding box. Particularly for hierarchical selection methods, determining whether or not a bounding box overlaps with a geometric selector can lead to early-termination of some selection operations.
- `select_bbox_edge(lower_left, upper_right)` : This is a special-case of the bounding box routine that provides information as to whether or not the *entire* bounding box is included or just a *partial* portion of the bounding box.

We demonstrate a handful of selection operations on a low-resolution dataset below. In Figure [3](#) we illustrate the selection of a rectangular prism (i.e., a `region` , like in Section [1.6.2.17](#). In Figure [4](#), we illustrate the selection of a sphere (i.e., a `sphere` , like in Section [1.6.2.20](#). And, to demonstrate yt's ability to construct boolean selectors from these objects (i.e., Section [1.6.2.2](#) we show what the logical `NOT` of these two objects would produce in [5](#).
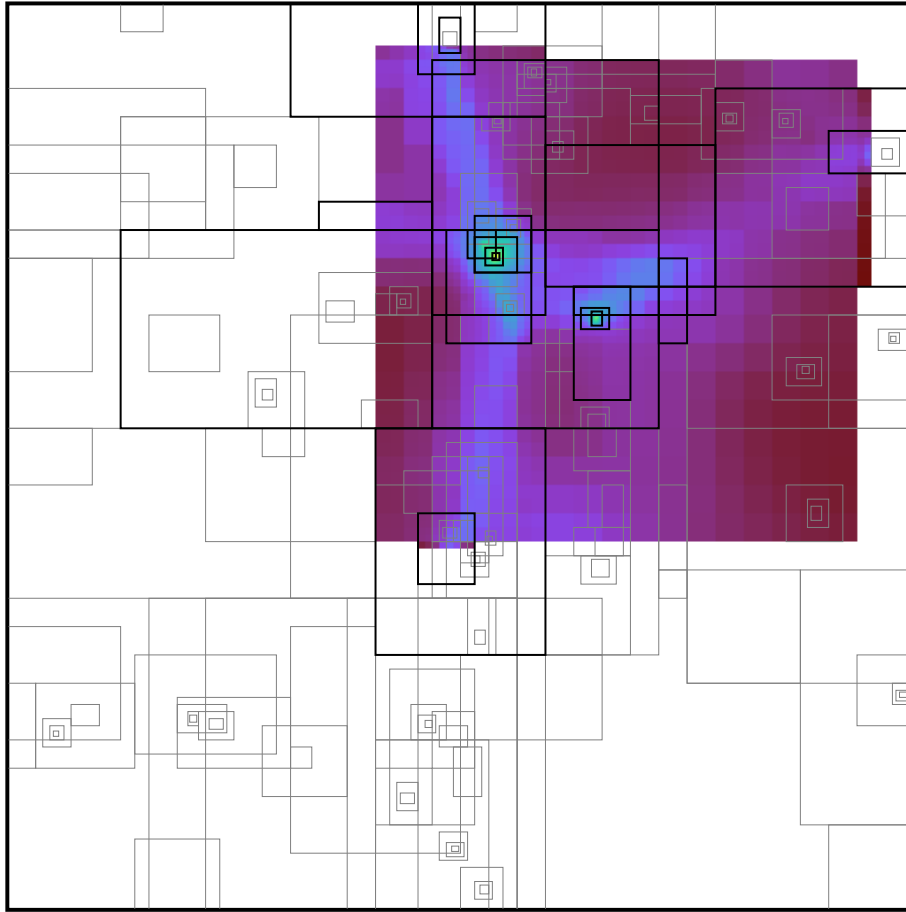
Figure 3: A selection of data in a low-resolution simulation from a rectangular prism.
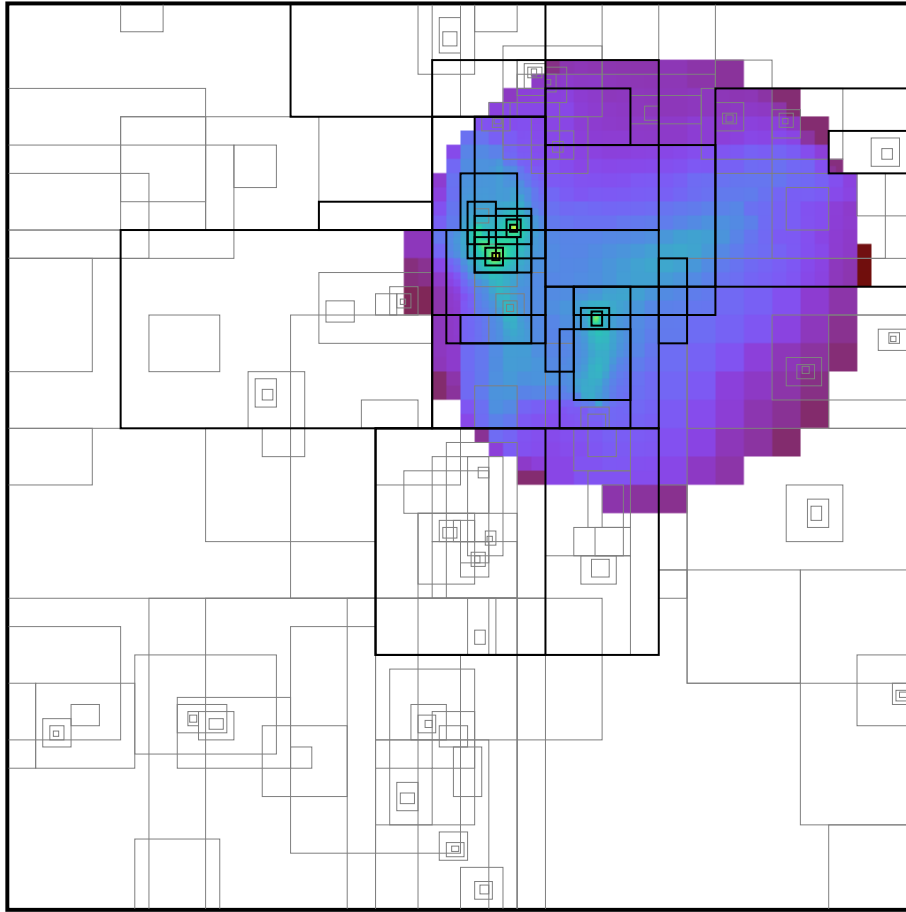
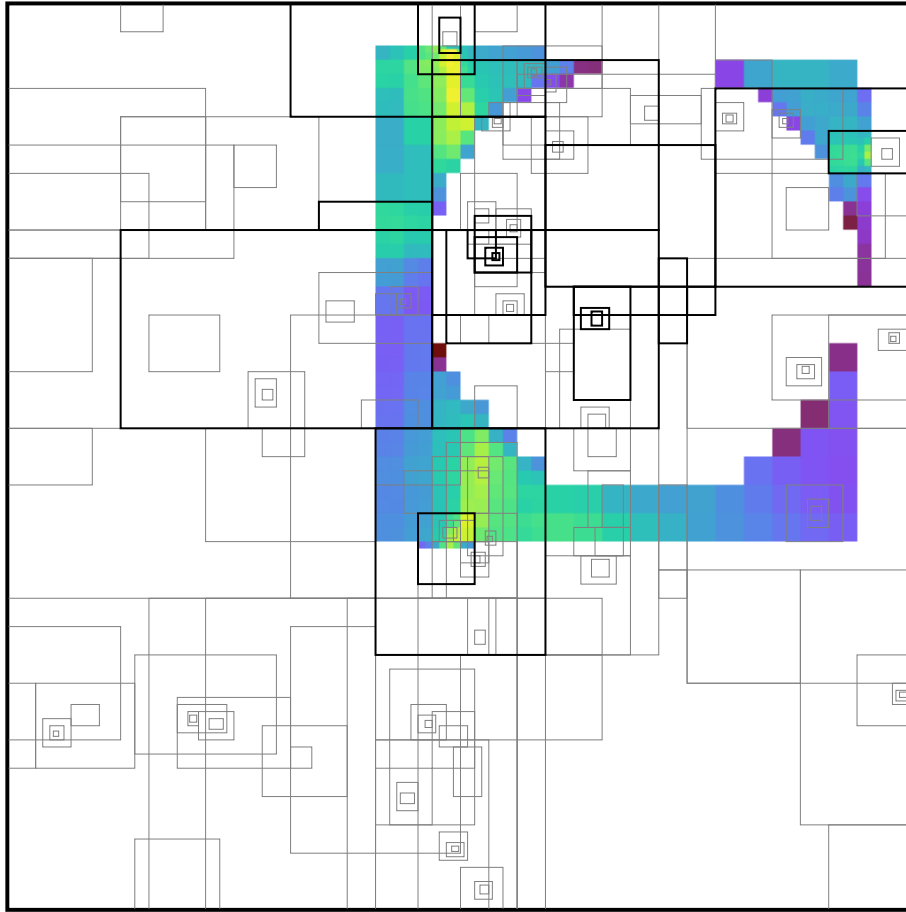**Figure 4:** A selection of data in a low-resolution simulation from a sphere.

Figure 5: The logical `A AND NOT B` for regions `A` and `B` from Figures [3] and [4] respectively.

Fast and Slow Paths

Given an ensemble of objects, the simplest way of testing for inclusion in a selector is to call the operation `select_cell` on each individual object. Where the objects are organized in a regular fashion, for instance a "grid" that contains many "cells," we can apply both "first pass" and "second pass" fast-path operations. The "first pass" checks whether or not the given ensemble of objects is included, and only iterates inward if there is partial or total inclusion. The "second pass" fast pass is specialized to both the organization of the objects *and* the selector itself, and is used to determine whether either only a specific (and well-defined) subset of the objects is included or the entirety of them.

For instance, we can examine the specific case of selecting grid cells within a rectangular prism. When we select a "grid" of cells within a rectangular prism, we can have either total inclusion, partial inclusion, or full exclusion. In the case of full inclusion, where the entire grid is included within the selector, we simply sidestep the specific inclusion checks completely and return a full mask of cells to utilize. In the case of partial inclusion, we can often determine the "start" and "end" indices of inclusion in the rectangular prism by examining the intersection volume. This allows us to avoid many costly individual `select_cell` calls.

With discrete point selection (and for our purposes, often unstructured mesh falls into this category) we often do not have the same organizing principle on which we can rely. However, utilizing hierarchical bitmap indexing we can often organize subsets of particles into collections of cells which may or may not be contiguous. In this situation, we can check for full inclusion within data objects, although we are not able to identify start and stop indices as the data are not assumed to be organized spatially independent of how we have indexed them.

At present, the objects listed in [2] are provided as selectors in yt. We do make a distinction between "selection" operations and "reduction" or "construction" operations (such as projections and smoothing/resampling), but have included both here for consistency. Additionally, some have been marked as not "user-facing," in the sense that they are not expected to be constructed directly by users, but instead are utilized internally for indexing purposes. In columns to the right, we provide information as to whether there is an available "fast" path for grid objects.

Table 2: Selection objects and their types.

Object Name	Object Type
Arbitrary grid	Resampling
Boolean object	Selection (Base Class)
Covering grid	Resampling
Cut region	Selection
Cutting plane	Selection
Data collection	Selection
Disk	Selection
Ellipsoid	Selection
Intersection	Selection (Bool)
Octree	Internal index
Orthogonal ray	Selection
Particle projection	Reduction
Point	Selection
Quadtree projection	Reduction
Ray	Selection
Rectangular Prism	Selection
Slice	Selection
Smoothed covering grid	Resampling
Sphere	Selection
Streamline	Selection
Surface	Selection
Union	Selection (Bool)

Arbitrary grid

Arguments:

- Left edge
- Right edge
- Active Dimensions

A 3D region with arbitrary bounds and dimensions. In contrast to the Covering Grid, this object accepts a left edge, a right edge, and dimensions. This allows it to be used for creating 3D particle deposition fields that are independent of the underlying mesh, whether that is yt-generated or from the simulation

data. For example, arbitrary boxes around particles can be drawn and particle deposition fields can be created. This object will refuse to generate any fluid fields.

Bool

Arguments:

- Operation
- Data object 1
- Data object 2

This is a boolean operation, accepting AND, OR, XOR, and NOT for combining multiple data objects. This object is not designed to be created directly; it is designed to be created implicitly by using one of the bitwise operations (&, |, ^, ~) on one or two other data objects. These correspond to the appropriate boolean operations, and the resultant object can be nested.

Covering grid

Arguments:

- Level
- Left edge
- Active Dimensions

A 3D region with all data extracted to a single, specified resolution. Left edge should align with a cell boundary, but defaults to the closest cell boundary.

Cut region

Arguments:

- Base object
- Conditionals

This is a data object designed to allow individuals to apply logical operations to fields and filter as a result of those cuts.

Cutting

Arguments:

- Normal
- Center

This is a data object corresponding to an oblique slice through the simulation domain. This object is typically accessed through the `cutting` object that hangs off of index objects. A cutting plane is an oblique plane through the data, defined by a normal vector and a coordinate. It attempts to guess an 'north' vector, which can be overridden, and then it pixelizes the appropriate data onto the plane without interpolation.

Data collection

Arguments:

- Object List

By selecting an arbitrary *object_list*, we can act on those grids. Child cells are not returned.

Disk

Arguments:

- Center
- Normal vector
- Radius
- Height

By providing a *center*, a *normal*, a *radius* and a *height* we can define a cylinder of any proportion. Only cells whose centers are within the cylinder will be selected.

Ellipsoid

Arguments:

- Center
- a
- b
- c
- e0
- tilt

By providing a *center,A,B,C,e0,tilt* we can define a ellipsoid of any proportion. Only cells whose centers are within the ellipsoid will be selected.

Intersection

Arguments:

- Data objects

This is a more efficient method of selecting the intersection of multiple data selection objects. Creating one of these objects returns the intersection of all of the sub-objects; it is designed to be a faster method than chaining & ("and") operations to create a single, large intersection.

Minimal sphere

Arguments:

- Center
- Radius

Build the smallest sphere that encompasses a set of points.

Octree

Arguments:

- Left edge
- Right edge
- Particle count refinement criteria

A 3D region with all the data filled into an octree. This container will mean deposit particle fields onto octs using a kernel and SPH smoothing.

Ortho ray

Arguments:

- Axis
- Coords

This is an orthogonal ray cast through the entire domain, at a specific coordinate. This object is typically accessed through the `ortho_ray` object that hangs off of index objects. The resulting arrays have their dimensionality reduced to one, and an ordered list of points at an (x,y) tuple along `axis` are available.

Particle proj

Arguments:

- Axis
- Field
- Weight field

A projection operation optimized for SPH particle data.

Point

Arguments:

- P

A 0-dimensional object defined by a single point

Quad proj

Arguments:

- Axis
- Field
- Weight field

This is a data object corresponding to a line integral through the simulation domain. This object is typically accessed through the `proj` object that hangs off of index objects. YTQuadTreeProj is a projection of a `field` along an `axis`. The field can have an associated `weight_field`, in which case the values are multiplied by a weight before being summed, and then divided by the sum of that weight; the two fundamental modes of operating are direct line integral (no weighting) and average

along a line of sight (weighting.) What makes `proj` different from the standard projection mechanism is that it utilizes a quadtree data structure, rather than the old mechanism for projections. It will not run in parallel, but serial runs should be substantially faster. Note also that lines of sight are integrated at every projected finest-level cell.

Ray

Arguments:

- Start point
- End point

This is an arbitrarily-aligned ray cast through the entire domain, at a specific coordinate. This object is typically accessed through the `ray` object that hangs off of index objects. The resulting arrays have their dimensionality reduced to one, and an ordered list of points at an (x,y) tuple along `axis` are available, as is the `t` field, which corresponds to a unitless measurement along the ray from start to end.

Region

Arguments:

- Center
- Left edge
- Right edge

A 3D region of data with an arbitrary center. Takes an array of three *left_edge* coordinates, three *right_edge* coordinates, and a *center* that can be anywhere in the domain. If the selected region extends past the edges of the domain, no data will be found there, though the object's `left_edge` or `right_edge` are not modified.

Slice

Arguments:

- Axis
- Coord

This is a data object corresponding to a slice through the simulation domain. This object is typically accessed through the `slice` object that hangs off of index objects. Slice is an orthogonal slice through the data, taking all the points at the finest resolution available and then indexing them. It is more appropriately thought of as a slice 'operator' than an object, however, as its field and coordinate can both change.

Smoothed covering grid

Arguments:

- Level
- Left edge
- Active Dimensions

A 3D region with all data extracted and interpolated to a single, specified resolution. (Identical to covering_grid, except that it interpolates.) Smoothed covering grids start at level 0, interpolating to fill the region to level 1, replacing any cells actually covered by level 1 data, and then recursively repeating this process until it reaches the specified `level`.

Sphere

Arguments:

- Center
- Radius

A sphere of points defined by a *center* and a *radius*.

Streamline

Arguments:

- Positions

This is a streamline, which is a set of points defined as being parallel to some vector field. This object is typically accessed through the Streamlines.path function. The resulting arrays have their dimensionality reduced to one, and an ordered list of points at an (x,y) tuple along `axis` are available, as is the `t` field, which corresponds to a unitless measurement along the ray from start to end.

Surface

Arguments:

- Data source
- Surface field
- Field value

This surface object identifies isocontours on a cell-by-cell basis, with no consideration of global connectedness, and returns the vertices of the Triangles in that isocontour. This object simply returns the vertices of all the triangles calculated by the [marching cubes](#) algorithm; for more complex operations, such as identifying connected sets of cells above a given threshold, see the extract_connected_sets function. This is more useful for calculating, for instance, total isocontour area, or visualizing in an external program (such as [MeshLab](#).) The object has the properties .vertices and will sample values if a field is requested. The values are interpolated to the center of a given face.

Union

Arguments:

- Data objects

This is a more efficient method of selecting the union of multiple data selection objects. Creating one of these objects returns the union of all of the sub-objects; it is designed to be a faster method than chaining | (or) operations to create a single, large union.

Processing and Analysis of Data

`yt` provides several interfaces for accessing the data available in a given dataset. As described in [1.6](#), the primary means of accessing data is through "data objects" that apply selections to the dataset. These objects present dictionary-like interfaces that return data; below, we describe what options are available for the data that is returned ([1.7.1](#)), as well as high-level interfaces for applying aggregations and reductions ([1.7.2](#)).

Field System

In `yt`, there are three types of "fields" that define values at a given spatial location. The first of these is an "on-disk" field, representing the raw, unmodified (except potentially up-cast to 64 bit precision) values read from the data storage that defines the dataset, such as files or bucket storage; while `yt` does provide routines for reading these fields, they are passed largely unmodified and so we do not discuss them in depth. The second type of field is a "derived field," which is a functional definition of how to process or combine one or more fields that exist in the dataset. Finally, providing the closure necessary for these derived fields to be accessed independently of their naming convention are "alias fields" that provide mappings between platform- or format-specific names for fields and those used internally in `yt`. This collection of three classes of fields

Fields are also defined by their "sampling type" to distinguish between those fields defined in a volume-filling fashion (i.e., cell-based fields) and those that are defined by discrete samples that may or may not require closure or convolution functions to be applied. Fields that are defined as a collection of discrete samples can be combined or filtered differently than those that are defined in a volume-filling manner, as described in [1.7.1.3](#) and [1.7.1.4](#).

Field Aliases

Small differences in naming fields can prove disproportionately challenging for writing platform-neutral analysis code. For instance, if one platform names the "density" field `dens` and another refers to it as `Density` (or, as we have seen in one platform, even the unicode character for ρ) then any platform-independent derived field that utilizes density must be defined multiple times to refer to this fundamentally identical quantity. (An important note here is that in many cases, the reverse problem is true – some codes may refer to things with the same name but with different underlying definitions, which provides an additional challenge to the analysis process by requiring [disambiguation](#).)

To address this issue, `yt` defines a set of fundamental fields, along with a naming convention for extensibility, that are provided as "aliases" for the dataset-specific field names. This enables a consistent ontology to be defined for fields in `yt`, upon which the remainder of derived fields can rely. Typically these are defined by the authors of a given dataset format frontend, wherein a translation or lookup table is provided to match the on-disk fields to those expected by `yt`.

In some cases, it is through a combination of derived and aliased fields that the full set of data is made available to the researcher; for instance, some datasets do not store velocity as a quantity on disk, but instead store momentum. In this case, momentum is aliased from the on-disk field to the `yt` field, and then a derived field is generated to seamlessly provide access to the velocity field wherever it is needed.

Derived Fields

In addition to the fields that are defined in the dataset, `yt` recognizes that there exist essentially infinite fields *in potentia* that can be defined. For instance, commonly in astrophysics datasets the "density" of different elemental abundances are stored (which provides a natural conservation scheme

with the density) in the dataset. A simple derived field might be defined to provide the "fraction" of a given field:

$$f_X \equiv \frac{\rho_X}{\rho}$$

`yt` provides the ability to define this as a derived field in a functional form. For instance, if the density of helium is stored as the field-tuple `("gas", "helium_density")` we can define the function as:

```python
def _helium_fraction(field, data):
    return data["gas", "helium_density"] / data["gas", "density"]
```

Note that here, the argument `field` is a field definition object and `data` is a data object which we are using for our selection. This is the form that derived fields in `yt` take; these can be supplied to the function `add_field` (or they can use `derived_field` as a decorator) and they will become available for all data objects.

These fields can accept parameters (associated with the base data object used for selection) and can require that spatial information is made available to the derived field; this can enable the calculation of finite-difference stencils for operations such as averaging and operators such as the gradient.

Derived fields are an extremely integral component of `yt` and are the gateway to enabling low-memory overhead calculations and sharing of analysis code. In addition, `yt` includes a large number of fields available, many of which are dynamically constructed according to metadata available in the dataset, to jump-start analysis. Researchers using `yt` can load a dataset and immediately compute, for instance, the velocity divergence and `yt` will construct the appropriate finite different stencil, fill in any missing zones at the edge of individual boundaries, and return an array that can be accessed, visualized or processed.

Particle Filters

Many of the data formats that `yt` accepts define particles as mixtures of a single set of attributes (such as position, velocity, etc) and then a "type" – for instance, intermingling dark matter particles with "star" particles. Where simulations are concerned, this can produce much more efficient code; since particles are typically evolved in the same fashion, storing them adjacent in memory can speedup operations such as time evolution steps. However, when reading the data in, they often need to be handled in fundamentally different ways. The analysis of dark matter particles in a galaxy, for instance, needs to be conducted differently than the analysis of collisional particles, or particles that arise from other phenomena (such as gas). `yt` provides a method for creating new "particle types" on the fly and applying existing derived fields to them. By adding a new "filter" method, particles that meet this criteria ("high-mass Black Holes," for instance, or "star clusters more than 1 billion years old") are accessible in a new field tuple. This enables all existing memory-conservative operations to act on them.

This filter, for example, checks and returns only those particles whose field `particle_type` is set to a value of 2.

```python
@yt.particle_filter(requires=["particle_type"], filtered_type="all")
def stars(pfilter, data):
    filter = data[(pfilter.filtered_type, "particle_type")] == 2
    return filter
```

In this case, `yt` also infers the name of the newly filtered type from the name of the function, and they become stars. Now all existing operations will work on field-tuples beginning with `"stars"` as their field type.

Particle Unions

The opposite operation to that in 1.7.1.3 is also accessible, by which multiple particle types can be combined and viewed as a single logical type. For instance, if "star particles" and "black hole" particles are distinct in a simulation of galaxy formation, they can be combined into a logical union:

```
u = ParticleUnion("massive_objects", ["bh", "stars"])
ds.add_particle_union(u)
```

Since unions are restricted to combinations *in full* of different types, their creation requires only specification of the particle types to combine. The set of available fields is the intersection of the fields available for *all* the combined types. If both particle types share fields A and B but only one shares C, the union will only have fields A and B accessible to it.

Field Detection

`yt` determines at dataset instantiation time the fields that are available to be computed. This provides the ability for researcher to *query* what fields are available, and additionally as a side-effect it provides information to the `yt` IO routines which fields need to be computed for a given derived field. By utilizing this information, `yt` can "resolve" all required fields when a derived field is requested. As such, it is able to identify that `("gas", "velocity_divergence")` relies on the velocity fields along each axis. If these are the fields that exist in the dataset, the resolution process concludes here. If, however, they need to be computed from the momentum and density fields, those become the fields that are read from the dataset.

This resolution of field dependencies enables `yt` to read only the fields that are necessary *and* to do so in a single pass over a file, reducing the initialization and seeking time within a file. Particularly in environments where metadata operations (required for an `open` system call) or seek operations (where dataset chunks may need to be looked up within a file as indexed by a header) are expensive, this can have significant impact on the overall performance, and by operating on a chunk-by-chunk basis, it further reduces the need to store multiple fields in memory simultaneously.

This computation does, however, come with an overhead. Detecting the fields that are required (and thus determining which fields are available) can be expensive, as many small sympy objects are created in the unit handling subsystem and many redundant calculations performed in the `yt`-specific field resolution code. This is an area of great interest for future optimizations, as the current situation benefits the access of large derived fields over iteration over many small datasets. In particular, an enormous amount of time in the unit testing framework is spent detecting fields for datasets that are only used once and then discarded.

Array-like Operations

In `yt`, a newly-constructed data selector contains no data – this enables data selectors for large regions, in extremely large datasets, to be lightweight and cheap to construct. By ensuring that these objects don't immediately consume resources, they can be manipulated and operated on in a high-level fashion, without taxing the computational power. While these data objects *can* return the full set of data they include, `yt` also provides array-like operations that do not require immediate access to the

full set of numerical values, and which align with the mental-model for data processing that `yt` exposes. As an example, consider the following two operations:

```
dd = ds.all_data()
dd["gas", "density"].max()
```

and

```
dd = ds.all_data()
dd.max(("gas", "density"))
```

Both are available in `yt`. As a side-effect of Python's object model, the first will access the `("gas", "density")` item in the object `dd`, itself a concatenated numpy array, and then execute the `max` method on it. The second will call the `max` method on the data object, supplying to it the name of the field. This allows `yt` to decide how to decompose, parallelize and process the data in a memory-efficient way, and spread across multiple processors. Additionally, by emphasizing that the "maximum" is being taken on the data object, rather than the numerical data, other operations can be exposed that build on the underlying data organization. For instance, taking the maximum along a given (spatial) axis:

```
sp = ds.sphere("center", (10.0, "m"))
sp.max(("gas", "temperature"), axis="z")
```

This translates our meaning – find the maximum value along the z-axis – into a dimensionality reduction operation that uses `yt`'s built-in "projection" method. These operations, on data objects (rather than the underlying arrays of values that are accessible through them) provide dataframe-like methods for querying very large, spatially registered data.

The array-like operations utilized in `yt` attempt to map to conceptually similar operations in numpy. Unlike numpy, however, these utilize `yt`'s dataset-aware "chunking" operations, in a manner philosophically similar to the chunking operations used in dask. Below, we outline the three classes of operations that are available, based on the type of their return value.

Reduction to Scalars

Traditional array operations that map from an array to a scalar are accessible utilizing familiar syntax. These include:

- `min(field_specification)`, `max(field_specification)`, and `ptp(field_specification)`
- `argmin(field_specification, axis)`, and `argmax(field_specification, axis)`
- `mean(field_specification, weight)`, `std(field_specification, weight)`, and `sum(field_specification)`

In addition to the advantages of allowing the parallelism and memory management be handled by `yt`, these operations are also able to accept multiple fields. This allows multiple fields to be queried in a single pass over the data, rather than multiple passes. Additionally, the `min` and `max` operations will automatically cache the results during a single pass, which means that calling `max` immediately after `min` (and vice versa) on the same data object and field will not require a recomputation.

In the case of `argmin` and `argmax`, the default returned "axis" will be the spatial coordinates of the minimum or maximum field value (respectively).

However, by specifying an axis or set of axes that correspond to fields, the field values will be queried at these minimum or maximum points. This allows, for instance, to query the value of "density" at the minimum "temperature." The operations `mean` and `sum` are available here in a non-spatial form, where they simply compute the scalar reduction independent of the spatial registration of the dataset.

Reduction to Vectors

- `profile(axes, fields, profile_specification)`

The `profile` operation provides weighted or unweighted histogramming in one or two dimensions. This function accepts the axes along which to compute the histogram as well as the fields to compute, and information about whether the binning should be an accumulation, an average, or a weighted average. These operations are described in more detail in **reference profile section**.

Remapping Operations

- `mean(field_specification, weight, axis)`
- `sum(field_specification, axis)`
- `integrate(field_specification, weight, axis)`

These functions map directly to different methods used by the projection data object. Both `mean` and `sum`, when supplied a spatial axis, will compute a dimensionally-reduced projection, remapped into a pixel coordinate plane. Importantly, if the dataset is a finite-volume dataset (grid, octree, etc), the results of these operations will be a variable-resolution mesh, rather than a fixed resolution image buffer.

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 $\sim 10^7$ 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, several tentative explorations have been made into an implementation of a more sophisticated "grid visitors" indexing and selection method, drawing on the approach used by the oct-visitors (described [below](#)). These were an attempt to unify the selection methods between octrees and grids, to reduce the overall code duplication and implementation overhead. 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. However, while such an implementation may be possible, the previous attempts were stymied by performance and maintenance considerations for the grid code, in particular related to the masking of "child" zones in an efficient and straightforward manner. A spatial tree is constructed, wherein parent/child relationships are established between grids.

Octree Analysis

`yt` supports octree-based AMR datasets (primarily RAMSES and ART, but also the output from the octree-based radiative transfer code Hyperion). `yt` stores a copy of the octree using a pointer-based approach, where each oct points to its eight children (if refined). The octs living at the coarsest level of the simulation are stored as a uniform grid. For domain-decomposed datasets, each domain is represented as a sparse octree, where the root octs are stored as a list and efficiently accessed using a binary search, ensuring each root oct is found in $O(\log(N))$ time, where N is the number of root octs in the domain. Each oct is represented as structure that contains the on-file location of the oct (`file_ind`) and its in-memory location (`domain_ind`), the index of the domain it belongs to (`domain`) and a list of pointers to its children (up to eight in 3D). This requires at most 88 bytes per oct.

In order to load data within a given region, a two-step approach is followed. First, the cells within the region of interest, as described in [Data Objects](#) are selected. `yt` relies on an oct-visitor machinery combined with selection routines. The tree is recursively traversed depth-first starting from the root grid, following only those branches that may intersect with the selected region. At the tip of each branch, the up-to-eight leaf cells are visited. In a first pass, the number of selected cells is computed and in a second pass, the on-file location of their parent oct is stored. Second, `yt` relies on the on-file location obtained from the octree traversal to lazily read data from disk. This ensures that only the minimal amount of data is being read and is particularly efficient when accessing a region spanning a small number of domains and/or a small number of refinement levels.

Recently, `yt` has been extended to fully supports accessing neighboring cells. This is achieved by computing one-cell thick quantities around each oct, which emulates the "ghost zones" found in patch-based codes. This approach has the advantage of abstracting the octree structure and provides a common interface to create derived fields, as described in **CC: any section describing this?**. The 56 neighbors ($4^3 - 2^3$) surrounding each oct are found by performing a search in the octree, which finds any neighbor in $O(\mathrm{level})$, where level is the level of the central oct. The search is illustrated on Figure [6](#). Other optimizations are possible that trade computational time with memory, for example by storing the tree as a fully-threaded structure (i.e. store pointers to the 6 neighbors sharing a face with each oct), or by starting at a central oct and searching "upwards and outwards."
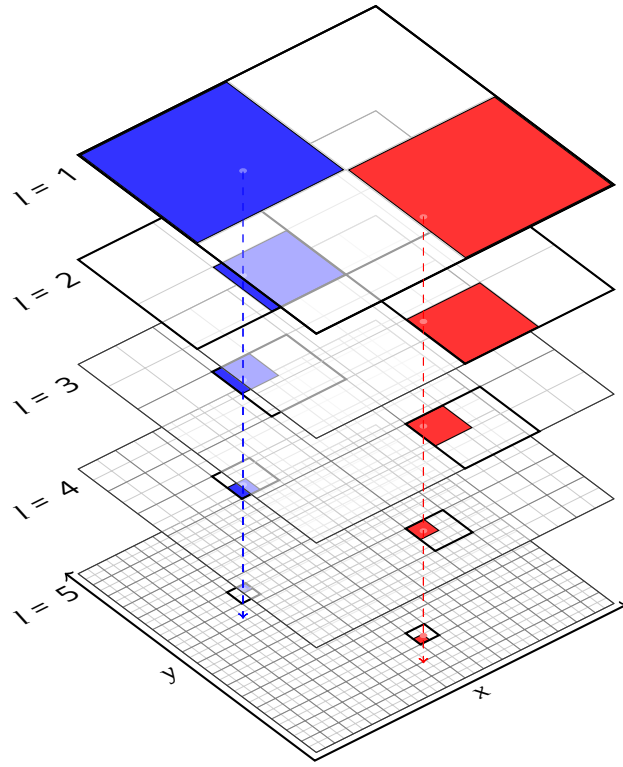
Figure 6: Illustration of a binary search through a quadtree. The search starts at the root level (level = 1 here) and recursively selects the quad that contains the point until reaching a leaf. The procedure is easily generalized in 3D.
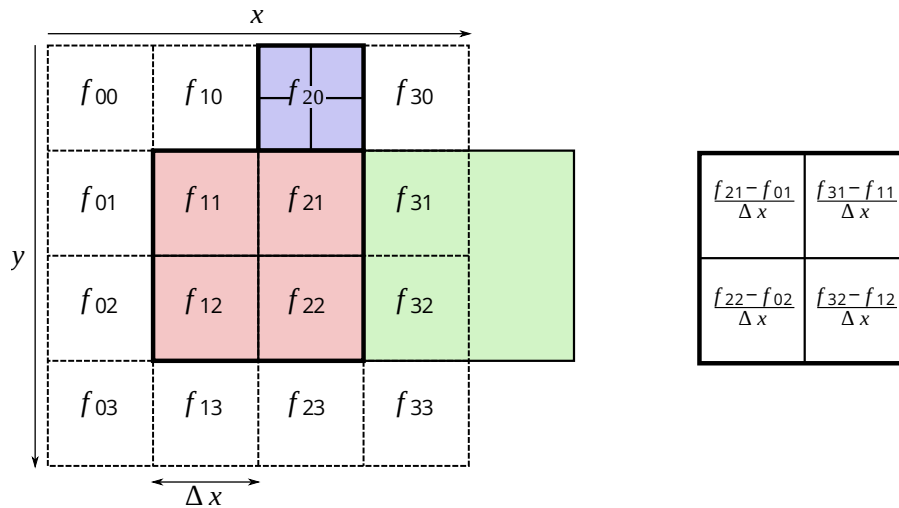

Figure 7: Scheme of the AMR structure used to estimate the gradient of a quantity in the central oct (red). Octs are represented in thick lines, cells in thin lines and virtual cells in dashed lines. *Left panel:* The virtual cell values on a 4^3 grid are interpolated from the nearest cell in the AMR grid. If the nearest cell is at the same (or coarser) level, its value is used directly. Note that virtual cells f_{31} and f_{32} have the value of the the actual coarser cell (green). If the cell is refined, the mean of its children is used (for example f_{20} is the mean of all the blue cells). *Right panel:* Gradients are estimated using a first-order finite difference centered scheme on the 4^3 virtual cells, here illustrated for a gradient along the x direction. **NOTE: the blue cell should be used in the example (for instance make it the f_{01} cell, rather than the f_{20} one since the y direction ends up not being used in the actual computation**

SPH Analysis

Smoothed Particle Hydrodynamics (SPH) is **MJT: We need a very brief explanation of SPH here!**

Previous versions of `yt` provided analysis of SPH data through a hybrid approach that mixed pure-SPH analysis with octree-based gridding and indexing that used particle density as a guide for the necessary resolution. Although the present, yt 4.0 series does not utilize octrees for particles, a description of the

previous implementation is useful to provide both historical information and modern motivation for the "demeshening" initiative that led to the current code base.

In practice, what this meant was that when a dataset was loaded, the particles positions were converted into one dimensional Morton index values, sorted, and by using a process of identifying the longest prefix in their bitwise representations, an octree (that mapped directly onto the Morton index values) was constructed. This octree was controlled by two parameters: `n_ref`, which specified the number of particles in an octree leaf node necessary to refine that node into eight sub-octants, and `over_refine_factor`, which specified the number of cells that each leaf node represented. For instance, if `n_ref` was set to 64 (the default), any octree node containing 64 particles would be refined into eight child nodes; if `over_refine_factor` was set to N, each leaf node would consist of a set of zones that were 2^N zones on a side (i.e., the default `over_refine_factor` produced eight mesh elements total).

Constructing these octrees using morton indices, if the entire set of particles could be stored in memory simultaneously, was *extremely* efficient. To do so, the particles merely needed to be converted into a morton index via fast, bit-level operations, those index values sorted, and then processed in order to identify the greatest common bit-prefix. Because two successive particles with identical index values would share an octree location, looking for sequences of identical prefix values (i.e., lower-level octree colocation) naturally produces an octree. When fluid quantities such as density were requested in the yt 3.0 series, the values were computed on the mesh defined by the octree; increasing the `over_refine_factor` and decreasing the `n_ref` would serve to increase the resolution. While this produced mostly-acceptable visualizations, and particular produced dynamically-resolved visualizations, it posed several problems for both visualization and analysis. The first, and arguably the most important, is that the strict locality requirements for refinement produced artifacts at leaf node boundaries. This resulted in incorrect and unphysical visualizations of hydrodynamic quantities, affecting most obviously those regions at the edges of clusters of gas particles. These were mitigated in regions of highly-clustered gas particles, but visual artifacts were still clear, as `yt` was applying a visualization suited for finite volume elements to Lagrangian particles.

With the 4.0 series, `yt` no longer utilizes octrees for analyzing, meshing or visualizing SPH data. Indexing, for the purposes of fast, memory-efficient access to subsets of the data, is provided by a bitmap index using the Morton indices of the particles, as described in [Bitmap Indexing](). For the purposes of visualization, any quantity requiring smoothing over nearest neighbors is computed on-demand at each pixel in the output image; this provides much higher resolution than the previous method, which was both subject to free parameters and required the construction of a 3D fluid field that was then collapsed to 2D for visualization. In many cases, this is also considerably more performant, as constructing a full-domain 3D fluid field is avoided, thus reducing both memory requirements and the number of floating point calculations.

Development of this new method was referred to internally as "the demeshening," as it served to eliminate the global (octree) mesh. In order to facilitate the massive, type and dimensionality-specific spatial queries necessary for performing millions of queries as efficiently as possible, and with as little overhead as possible, `yt` packages a kD-tree written in Cython that can be called from either Cython or Python, and which provides low-level APIs for querying from within tight loops. Whereas previously, constructing a projection or a slice would slice through an octree mesh and provide the results from that variable resolution mesh, the current version of `yt`'s SPH machinery will instead construct a pixel plane and smooth the appropriately identified particles onto that pixel plane. This produces much higher-fidelity results (see Figure [8]()), but a current limitation is that whenever the pixel plane is changed, the particles must be re-deposited; this puts it at odds with the similar machinery for octree and patch-based datasets, which provide a "read-once-pixelize-many" approach.

The octree method – while not incapable of utilizing different normalization and particle search methods – was less flexible than the current, de-meshened approach. For instance, the method of SPH particle identification (i.e., so-called "scatter" or "gather" methods for correlating particles with positions) is now flexible and able to be set at runtime. The normalization (if used) can take into account global quantities, local quantities, and is flexible based on the field being smoothed.
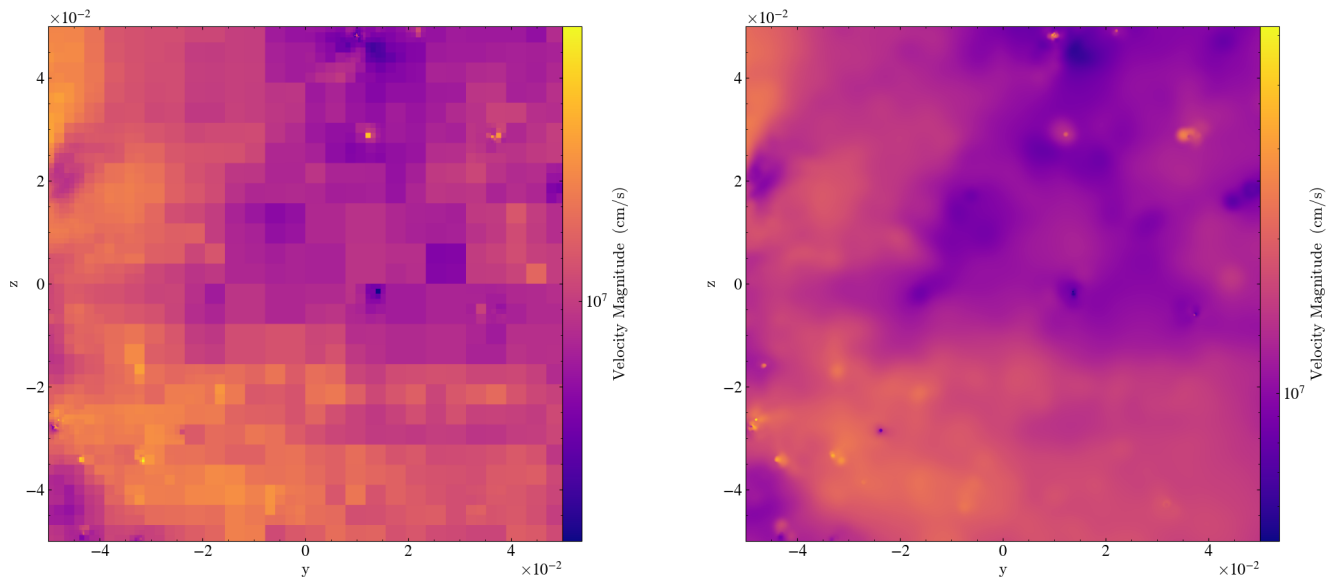

Figure 8: Comparisons between the older, octree-based method used in version 3.0 of `yt` (left) and the newer, "demeshened" algorithm used in yt 4.0 and beyond (right). The left image clearly shows artifacts from the octree structure imposed on the underlying dataset by `yt`, and the right hand side is much smoother, with more definition at individual pixels. The difference in color bars is notable as well, accounted for by the different normalization methods.

Some additional differences between SPH analysis and the analysis of finite volume data are present when utilizing data selectors. For instance, 3D data selectors as applied to finite volume codes only select those cells whose centers fall within the data selector. 2D and 1D data selectors (such as slices and rays) also include those finite volume cells that the selector passes through. However, with SPH data, the selection methods in 2D and 3D will always include those particles whose spheres of influence, defined by the appropriate smoothing lengths, are within or overlapping with the data selector. This is somewhat counter to the expectations set by the grid codes, but aligns with the need to have a fully self-contained data-container for computing field values. For instance, this means that a "ray" object (often used to compute, for instance, the column density in a cosmological simulation) will in fact include a set of particles within a (potentially) varying impact parameter.

MJT: Needs a diagram, could be drawn from the contrived test case

Other than these differences, which have been intentionally made to align the results with the expected results from the underlying discretization method, the APIs for access to particle data and finite volume data are identical, and they provide broadly identical functionality, where the disparities are typically in functionality such as volume rendering. This allows a single analysis script, or package (such as Trident), to utilize a high-level API to address both gridded and Lagrangian data while still ensuring that the results are high-fidelity and representative of the underlying methods.

Unstructured Mesh Analysis

Non-Cartesian Coordinates

Indexing Discrete-Point Datasets

Advances in both hardware and software facilitate astrophysical datasets of growing complexity and size. The datasets produced by numerical simulations can currently reach sizes of $$100 Tbytes split across hundreds of files [e.g. 4]. For even simple analysis tasks, the cost of incrementally reading datasets this large into memory is quite high. This problem is not limited to theoretical work. During operations the Large Synoptic Survey Telescope (LSST) will produce 15 Tbytes of data each night [5]. In order to analyze such large datasets, we need innovative techniques for quickly indexing and selecting data without loading the entire dataset into memory. We present a technique for using Morton bitmap indexes to map files and accelerate data analysis.

Theory and Background

Domain Partitioning Between Files

A common analysis task is the selection of data within a subset of the full domain; we use the term "selector" to refer to the selection operator. If the dataset is split across multiple files, either due to size constraints or to allow for parallel I/O, such selections require every file to be loaded and parsed in order to assemble all of the data within the selection criteria. This process can be very costly in terms of both the memory required to store the data and the time required to read each file. However, if the contents of the files are mapped in advanced, only the files touched by the selection will need to be loaded. This is particularly effective for partitioning schemes that are localized within the domain. If each file contains data that are localized to one part of the domain, selections of contiguous sub-sections within the domain will require fewer files to be loaded. Figure 9 shows four examples of possible partitions of a two-dimensional spatial domain split equally between 8 files.

a) Random   b) Sliced   c) Morton   d) Hilbert
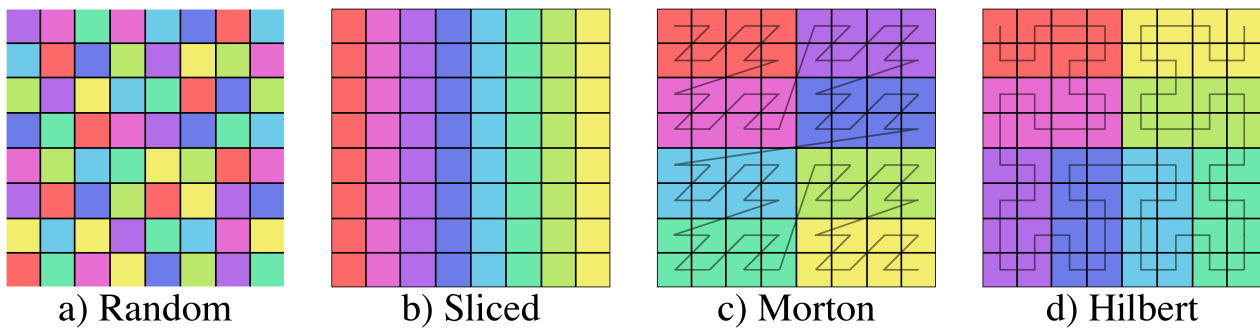
Figure 9: Examples of four different schemes for partitioning a 2D domain between 8 files. Each color represents a different file.

Panel (a) is an example where random parts of the domain are contained within each file. In such a case, many files will need to be loaded for contiguous selections within the domain. In panel (b), the domain was split between the files along the x dimension. Fewer files will need to be loaded for queries along the y-dimension, but contiguous selection in x will still require a greater number of files since the partition is not well localized in that dimension. Panels (c) and (d) are both examples of partitioning the domain between the files along a space filling curve [Morton and Hilbert curves respectively; [6]; [7]]. These partitions have the greatest chance of limiting the number of files that must be loaded for a contiguous selection with slightly improved localization for the Hilbert curve. Consequently, Hilbert curves have also been used for load-balancing in parallel simulation codes like Gadget-2 [8] and RAMSES [9].

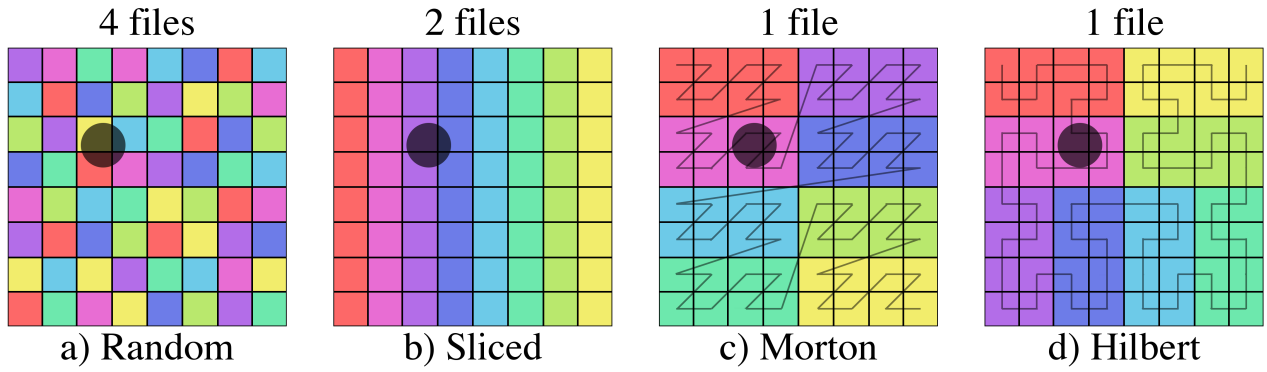Figure 10 shows examples of three selections within the above domain partitions.

Figure 10: Examples of file selection for four different domain partitions and three different shaded selectors. The number of files above each images is the number of files that must be loaded in order to get all of the data within the selected region.
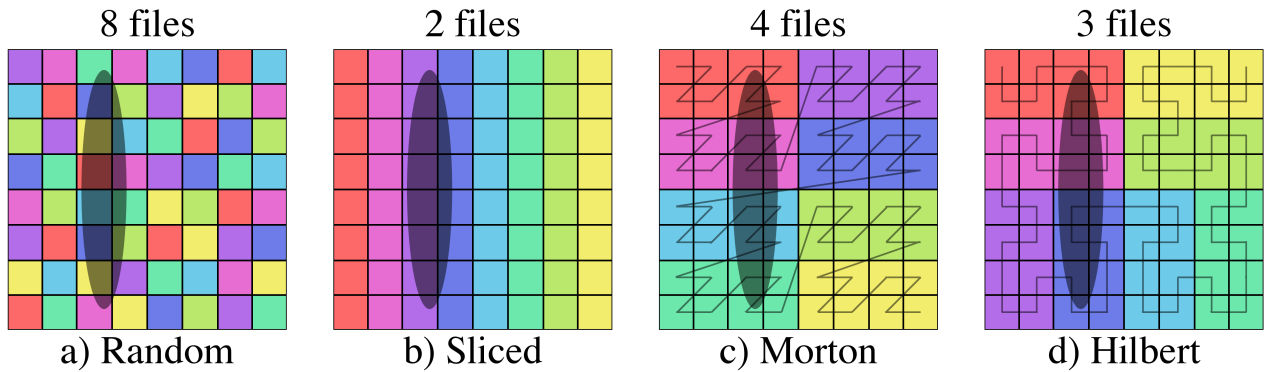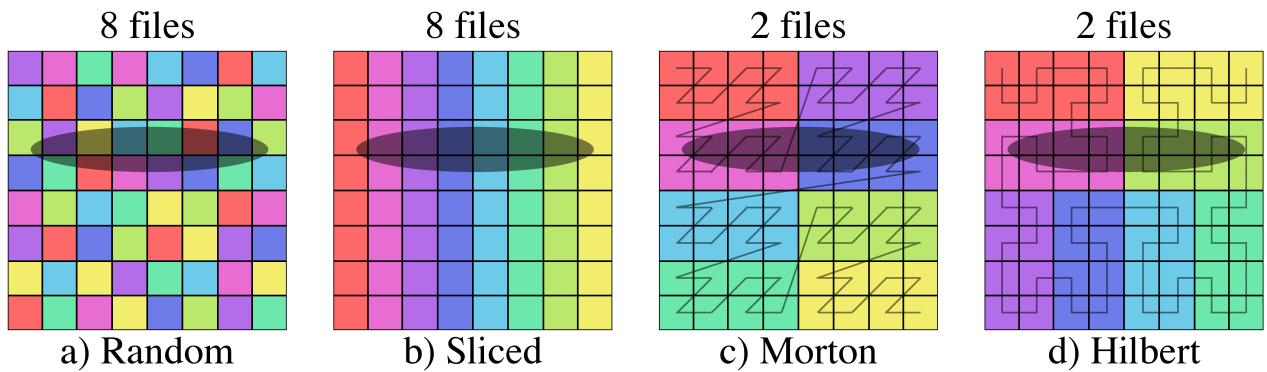

Figure 11: Examples of file selection for four different domain partitions and three different shaded selectors. The number of files above each images is the number of files that must be loaded in order to get all of the data within the selected region.

Figure 12: Examples of file selection for four different domain partitions and three different shaded selectors. The number of files above each images is the number of files that must be loaded in order to get all of the data within the selected region.

For the smallest selector (first row), the random domain decomposition (a) already requires half of the files to be loaded while more localized schemes require much fewer. Similarly, while the sliced domain partition (b), requires the fewest files to be loaded when the selector is oriented in the same direction as the slicing (second row), it requires *all* of the files when the selector is perpendicular to the slicing (third row). While some datasets may have information on the domain range covered by each file, the partitioning scheme used for simulation output is often decided at runtime, can be system dependent, and may be imperfect.

Files are often partitioned for parallel I/O such that each processor outputs data on the portion of the domain it is responsible for processing. To limit the cost of communication between processors, the domain will be split across processors such that neighboring processors are responsible for neighboring parts of the domain. This means that, although the overall partitioning scheme may be known for a given dataset, the exact order of the files will be dependent on the configuration of the processors at runtime.

The partitioning can also be imperfect if the domain decomposition is not perfect at the time of output. For instance, in astrophysical N-body simulations, it is possible for particles to travel from one processor's domain to another. In this case, the partition will only be perfect directly following an update to the domain decomposition.

In cases where the exact file organization is not known or imperfect, it is advantageous to map the files post-process in order to speed up selections for analysis. Although the same result can be achieved by re-sorting the data itself, creating the map can be less computationally less expensive than re-sorting the data, can be saved for use with multiple selections, and does not required write access; this is typically not feasible, especially in the case of datasets shared by large, distributed communities.

Morton Indices

Morton ordering maps multidimensional data onto a one-dimensional space filling curve [6]. This is done by breaking up the domain into cells where each cell's position within the N-dimensional domain can be described by N integers. The Morton index of the cell is then created by interleaving the bits of the N integers to create a single integer that fully describes the cell's position (see panel (b) Figure 13). As seen in panel (a) of Figure 13, ordering of the cells by their Morton indices forms a space filling Z-curve.
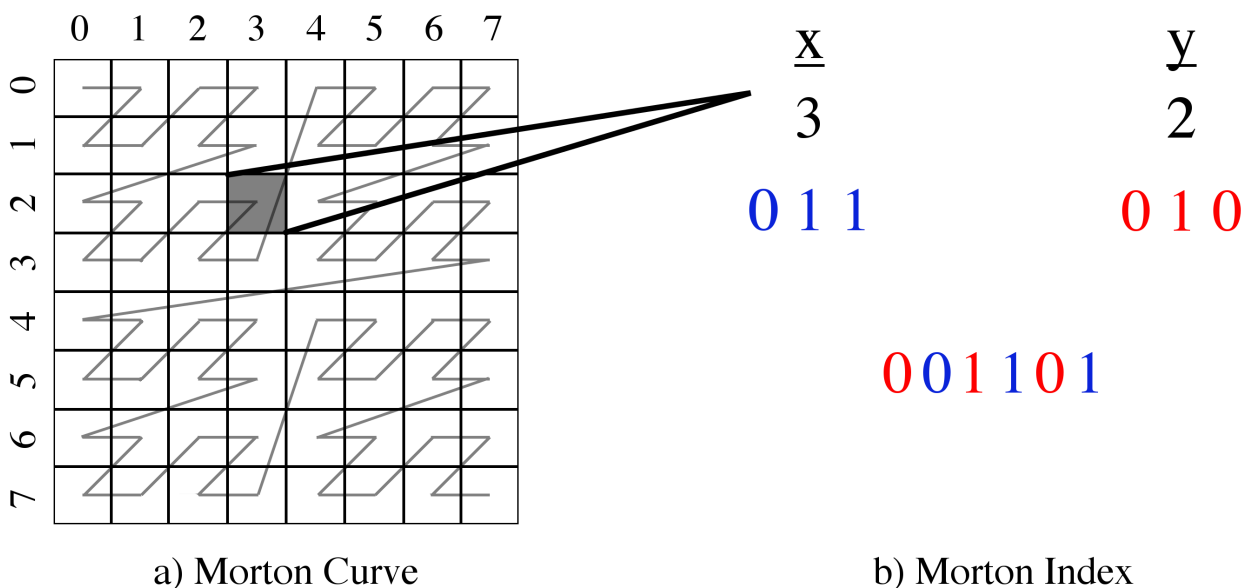

a) Morton Curve

b) Morton Index

Figure 13: Example of 3rd order Morton curve in two dimensions. The bits of the x and y indices are interleaved to generate a single integer that fully describes the cell's location within the two-dimensional domain to within $1/2^3$th of the domain in each dimension. **NOTE: wouldn't it make more sense to write y and x is this order, so that the interleaving could be represented with arrows without crossing them ?**

The precision of a single Morton index is only limited by the size of the integer used to store it. For instance, 64-bit Morton indices in 3 dimensions can be localized to $1/2^{21}$th of the domain in each

dimension (3×21 bits = 63 bits). If the domain is binarily divided into subcells to some order k in each dimension (i.e. 2^{Nk} cells), coarser Morton indices can be obtained by simply masking lower bits. Morton ordering has been used to speed up quadtree construction [10], nearest neighbor searches [11], and range queries [12]. By recording the indices of the cells containing data from each file within a dataset, Morton indices can also be used to construct one-dimensional maps of an N-dimensional dataset that can be represented as bitmaps.

Bitmaps & EWAH Compression

Bitmap indexes use the values of single bits within an array of bits to describe dataset properties. This form requires minimal memory and can be filtered using computationally inexpensive boolean operations. Bitmap indexes have long been popular for use with large data warehouses [13,14,15]. However, as scientific datasets have become larger and more complex, they have also begun to gain traction in a diverse array of scientific fields including geosciences [16], earth sciences *TODO: insert citation here ?*, rocket science [17,18], high-energy physics [19], and combustion [20].

In cases where data attributes can take on a finite set of values, one bitmap is constructed for each possible attribute value. Within the bitmap each bit specifies whether or not the corresponding data point has that value. In this way, queries for data with a single attribute value require consulting only one bitmap and queries of multiple attributes/values can be done using boolean AND operations on the corresponding bitmaps. In the case of scientific data, which often contains floating point value attributes, the attributes must be binned prior to constructing the bitmaps [21,22,23]. Here, Morton indices are used to bin N-dimensional floating point data onto one-dimension. As a result, each file can be described by one bitmap.

For each file within a dataset, the Morton indices touched by the data within that file can then be stored in a bitmap index for future searches where the value of bit j indicates whether or not Morton index j is touched by the file in question. For Morton indexing of order k, this would result in a bitmap of length 2^{Nk} bits per file. For large bitmaps, this can become costly in terms of memory and the time required to perform bitmap operations. However, Enhanced Word-Aligned Hybrid (EWAH) compression can be used to limit these costs, particularly when the domain is densely or sparsely populated in localized regions [24,25,26].

An EWAH compressed bitmap will be smaller when there are long sequences, or "runs," of identical values. This means that an EWAH compressed bitmap will be smallest if either all or none of its bits are set. An uncompressed bitmap would require the same, maximum, amount of memory in both of these cases. The locality of Morton indices takes advantage of the EWAH compression. If there are regions of the domain that are densely/sparsely populated, there Z-order space filling curve ensures that the bits denoting those regions will be adjacent, increasing the likelihood that there will be runs of identical (set/unset) bits and limiting the size of the compressed bitmaps.

Collisions

It is possible that two files will contain data within the same Morton cell. This would mean that any time that cell is touched by a selection, both files would need to be loaded even if the selection only touches data from one of the files. Figure 14 provides an example of collisions between two files. In panel (a) of Figure 14, purple cells are those that contain data from both files, a collision, for a 3rd order Morton index. Any selector that contained one of those cells would need to load all of the data from both files, even if it only selected part of the cell. Where the data is highly-concentrated in a central region (for instance, in a galaxy formation simulation with particles centrally-concentrated) this can mean that some regions suffer from worst-case scenario collision.
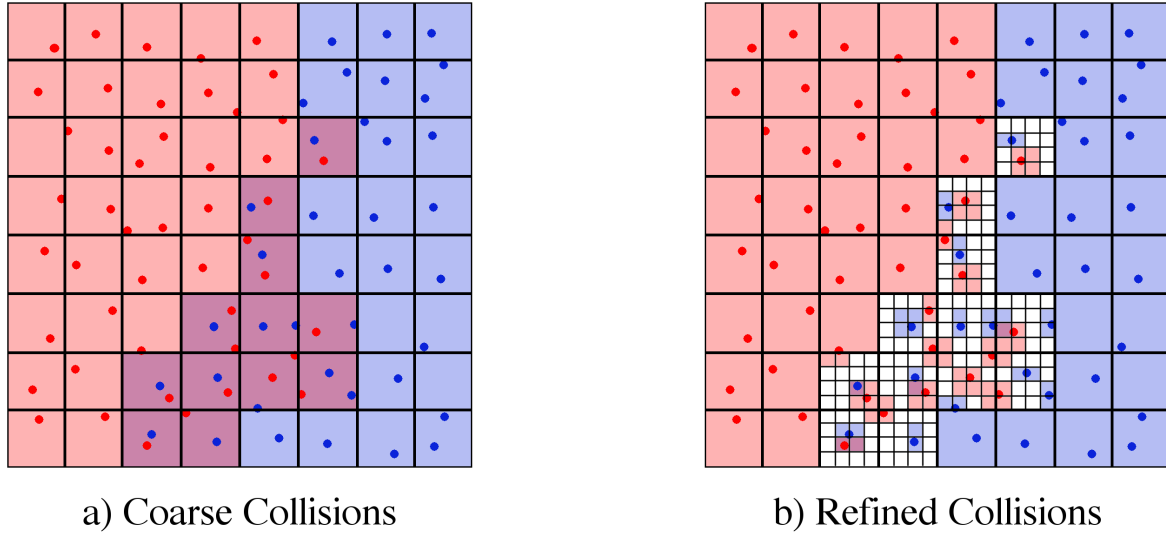
| a) Coarse Collisions | b) Refined Collisions |

Figure 14: Examples of a collision between two files. The red points and blue points are contained by two different files. The larger grid in both panels denotes the boundaries of 3rd order Morton cells. The cells containing points from either file are shaded accordingly such that cells containing points from both files are purple. The smaller grids within these cells on the right are the boundaries of 2nd order Morton cells refining the collisions.

Collisions can be limited by either increasing the order of the index or allowing for multi-resolution indexes [17,18]. Panel (b) of Figure 14 demonstrates an example of nesting a second index within cells that contain collisions. In those cells which contained collisions, a 2nd order Morton index was added. Those cells with collisions at the level of the refined index (purple cells in panel (b)) cover a much smaller portion of the domain than the cells with collisions at the level of the coarse index (purple cells in panel (a)). This means that any given selection is less likely to contain a collision and it will be less likely for a selector to require both files to be loaded unless it actually touches data from both files.

Increasing the order of the coarse index has the same effect as nesting a second refined index within cells with collisions, but can also increases the size of the resulting map and the time it takes to identify files touched by a selection. However, if the order of the coarse index is too small or the order of the refined index too large, this too can increase the cost of a selection in terms of memory and time. Section 1.10.2.1 discusses this tradeoff and how to choose index orders.

Collisions are more common for file partitioning schemes that are not localized. Figure 15 shows an example of collisions for the different partitioning schemes discussed in Section ??.
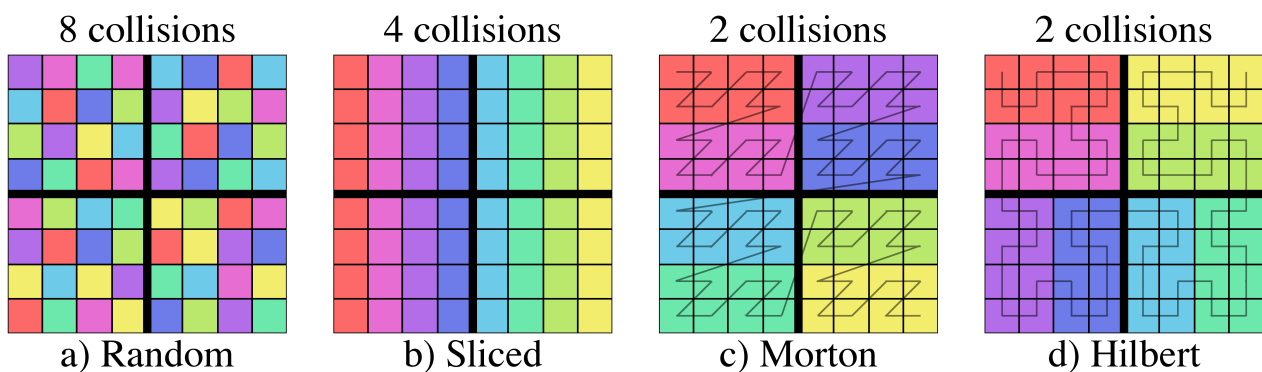

| 8 collisions | 4 collisions | 2 collisions | 2 collisions |
| a) Random | b) Sliced | c) Morton | d) Hilbert |

Figure 15: Examples of collisions for four different domain partitioning schemes. The heavy black lines denote 1st order Morton cells. The presence of more that one file (color) within a Morton cell indicates a collision. **NOTE: from an**

accessibility standpoint, I must note that using a smooth color gradient may be counter productive for colorblind readers as it makes it a lot harder to perveice that neighbouring regions belong to different files.

For the random domain partition in panel (a), every cell within a 1st order Morton index will contain data from all 8 files. This means that any selection using a 1st order bitmap index will require every file to be loaded. For the more localized partitions in panels (c) and (d), only two files touch each Morton cell.

Ghost Zones

It is often the case that, in selecting a region, additional padding around the region should be included in the selection. This is particularly useful for algorithms that need information about neighboring points in the domain [e.g. gas properties in simulations using Smoothed Particle Hydrodynamics; SPH; [27]; [28]; [8]]. For Morton indices, this is straightforward as the indices neighboring Morton cells can be found by incrementing the bits corresponding to each dimension. We have included the ability to pad selectors with some number of Morton cells referred to as 'ghost zones'. Those files that touch ghost zones, but not the selector itself are referred to below as 'ghost files'.

Depending on how the domain is split between files, the inclusion of ghost zones may or may not increase the number of files that need to be loaded. Figure 16 shows an example of a ghost zone around the first selector from Figure 10.
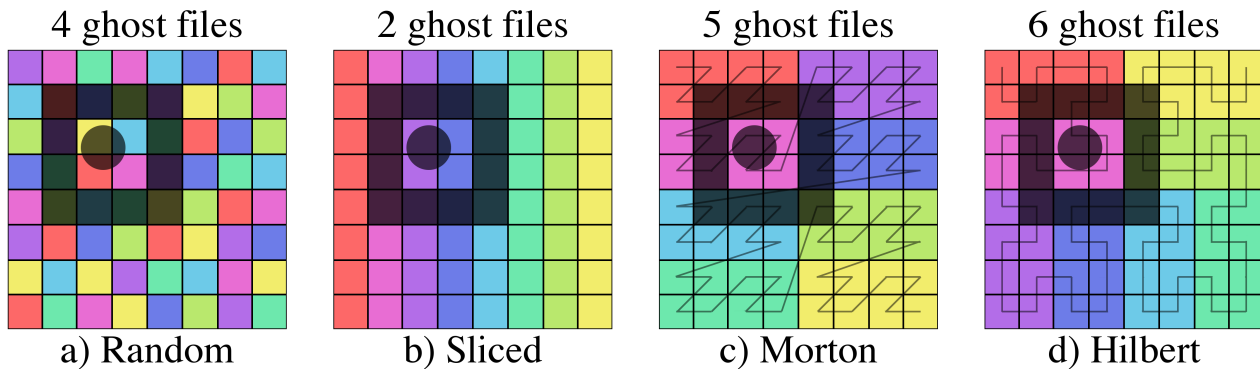
Figure 16: Examples of a selector ghost zone with a width of one Morton cell at an index order of 3 for four different domain partitioning schemes. The shaded circular region is the selector and the shaded box is the ghost zone. Different partitioning schemes will lead to different numbers of ghost files. **NOTE: same as figure 13**

The ghost zone has a width of one Morton cell at an index order of 3 and contains the same part of the domain in each case. However, due to differences in how the domain was partitioned between the files in the four cases, the number of additional ghost files touched by the ghost zone in each case is different. This will also depend on the order of the index to which ghost zones are added. Ghost zones added at the order of the coarse index will be larger than those added at the order of the refined index and will have a higher probability of touching additional files. While including ghost zones is advantageous when neighbor info is needed, it also increases the computational cost of identifying files (see Section 1.10.2).

Methods

The basic procedure for constructing the bitmap index is as follows:

1. **Compute coarse indices.** For each file in the data set, read in the data and compute the indices of Morton cells at a given coarse order that are touched by data contained within that file. These coarse indices are then stored by setting the corresponding bits in an EWAH compressed bitmap.

2. **Find collisions.** The indices of coarse cells that are touched by data in more than one file (collisions) are located using bitwise operations on the file bitmaps. These indices are also stored in an EWAH compressed bitmap.

3. **Compute refined indices.** For each file in the data set, read in the data and compute the indices of Morton cells at a given refined order within coarse cells with collisions that are touched by that file. These refined indices are stored in a map from coarse Morton index to an EWAH compressed bitmap of refined Morton indices within that cell.

4. **Output bitmaps.** The EWAH compressed bitmaps for the coarse indices, refined indices, and collisions are saved to an external index file.

For large datasets and/or high levels of refinement, this can be a time consuming process; however, it must only be done once. For future selections, the bitmap can be quickly loaded and used to identify files in less time than would be required to load and query each file within the dataset individually. Selection using a loaded bitmap goes as follows:

1. **Construct selector bitmap.** In the same way each file was mapped, the indices of Morton cells touched by the selector are stored in a bitmap. This is done by checking for intersection of the selector with Morton cells at the order of the coarse bitmaps. For contiguous selectors, this is done at lower order (parent) cells first and continued recursively until the order of the coarse bitmap is reached.

   - If a cell is completely within the selector, all of its child cells at the coarse order are added to the bitmap.

   - If a cell intersects the edge of the selector, child cells at increasing orders are checked until the order of the coarse bitmaps are reached. If the cell is at the coarse order and there is a collision between two files, a refined bitmap is the constructed for the selector in the same manner.

2. **Find files intersecting the selector.** Bitwise operations with the coarse file bitmaps are then used to efficiently identify files that intersect the selector within coarse cells. If the coarse cells within the intersection with a file all have collisions with other files, bitwise operations with the refined file bitmaps are then used to determine if the file is selected at the order of the refined index.

If ghost zones are desired, the neighbors of cells that intersect the edge of the selector are added to a separate bitmap. For cells without collisions, the neighbors are added at the coarse bitmap order. If there are collisions, the neighbors are added at the refined bitmap order.

Tests

The utility of using Morton index bitmaps for mapping files to decrease query times was tested on artificial N-body simulation datasets containing 1024^3 points in three dimensions, distributed between 512 files. For each test a Morton index bitmap was constructed for the dataset and used to identify files touched by cube shaped three-dimensional selectors. The performance is assessed in terms of the number of files identified and the average time required to identify them across 10 runs. If fewer files are touched, fewer files will need to be loaded during analysis of a selected region and the overall fraction of time spent on I/O will be lower. If less time is required to identify the files touched by a given selector, more selections can be made using the same computational resources. This was done for varying index orders (Section 1.10.2.1), selector sizes (Section 1.10.2.4), and partitions of the domain between files (Section 1.10.2.5).

Index Order

Overall Refinement

The order of the Morton indices used to map the files determines the time required to identify files and the number of collisions that will occur between files. Higher order indices will result in fewer collisions, but will take longer to query, as seen in Figure 17 Six selectors of varying sizes and positions within the domain where used to identify files based on Morton index bitmaps of varying order. The test dataset was split across the files using a Hilbert curve of order 6 with 10% scatter between Hilbert cells to simulate an imperfect domain decomposition as can occur if particle positions are updated and output prior to updating the domain decomposition.
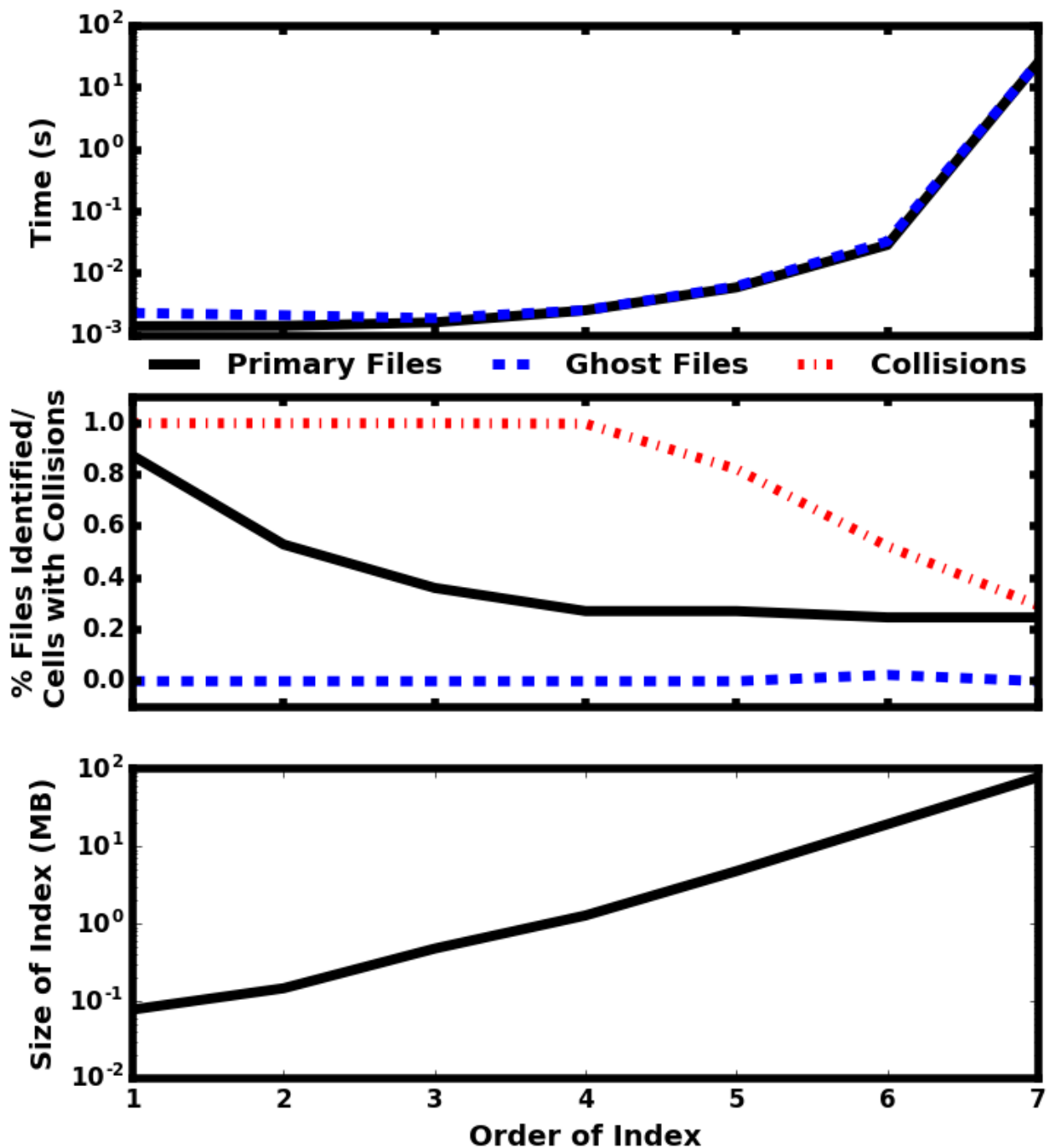
Figure 17: Dependence of query time (top), fraction of files selected/cells with collisions (middle), and index size (bottom) on the total refinement of the bitmap index. The solid black lines correspond to the query times and files identified by just

the selectors. The dashed blue lines correspond to the query times and additional files selected when a ghost zone with the width of one Morton cell is added around the selectors. The dash-dotted line in the middle panel shows the fraction of cells with collisions between files. **NOTE: maybe this would make more sense as scatter plots ? (connecting dots to fill a continuous yet unreachable space of non-integer-limited x values seems inappropriate.**

Below a bitmap index order of 4, there are collisions between multiple files within every cell, resulting in a larger number of files being identified. However, as the order increases, the number of collisions drops and the file count plateaus at $\sim 25\%$. This translates to a $\sim 75\%$ reduction in the memory and time required for processing files, a significant increase in performance. For a 7th order bitmap index, selection requires $> 100\times$ the time that the same selection took using a 6th order index, but there is no change in the number of files indicated. A 6th order index is sufficient to identify the minimal set of files touched by the selectors in this case because the dataset was partitioned between the files along a 6th order Hilbert space filling curve. While it is generally true that the time required to identify files using a bitmap index will increase exponentially with the size of the index, the order of the index that results in the minimal number of files for any dataset will depend upon how the domain is partitioned between files (see Section [1.10.2.5](#)). The memory required to store the index for the test dataset scales according to $\propto 2^{2k}$, for a kth order index. If uncompressed bitmaps had been used instead of EWAH compressed bitmap, the memory would have scaled with the total number of cells contained within the 3-dimensional test domain (2^{3k}).

Collision Refinement

Increasing the refinement of the primary index does so for the entire domain and, as seen in Section [1.10.2.2](#), can become costly in terms of the memory required to store the bitmap and the time required to perform operations. However, it is also possible to increase refinement by nesting a second Morton bitmap index within those cells of the primary index that contain collisions. As the nested indexes will contain a smaller portion of the domain and data, they will be less complex and can be compressed more efficiently than the primary index covering the entire domain. This enhanced compression means that, although a greater overall number of EWAH compressed bitmaps will need to be utilized (one for the coarse index and one for each collision within the coarse index), less space will be needed to store the bitmap and bitwise operations will be faster. Figure [18](#) shows the results for adding a secondary index of varying order with the overall refinement order of the index (primary index order + secondary index order) held constant at 6. The test dataset and selectors applied were the same as in Section [1.10.2.2](#).
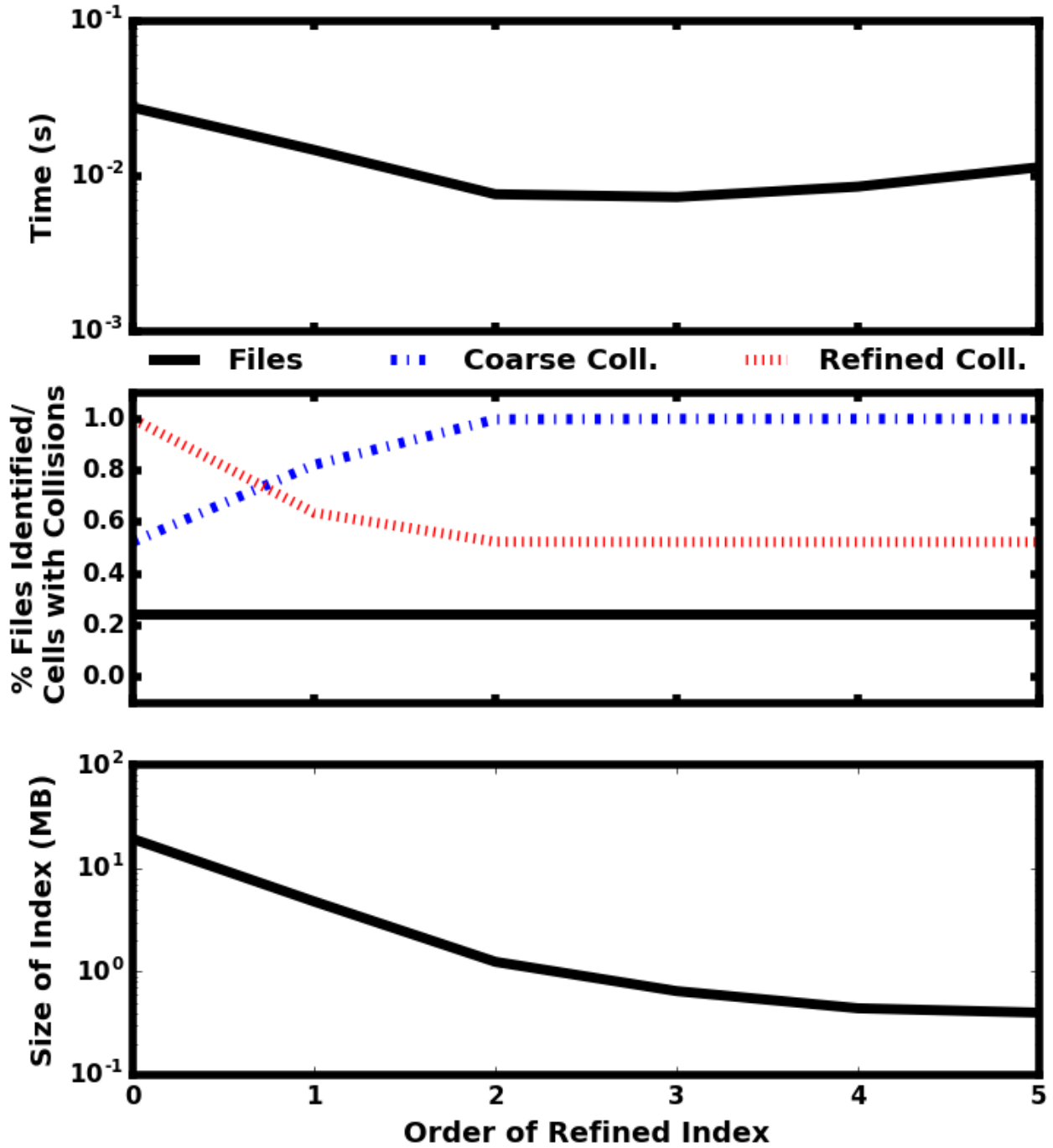
Figure 18: Dependence of query time (top), fraction of files selected/cells with collisions (middle), and memory required to store the index (bottom) on the order of the secondary index used to refine collisions. In the middle panel, the solid black line corresponds with the fraction of files identified, the dash-dotted blue line is the fraction of cells at the first index level that have collisions, and the dotted red line is the fraction of cells at the second index level that have collisions.

When the order of the second refined index is low, the first index is larger resulting in fewer cells with collisions at the first index and more at the second. The reverse is true when the order of second index is higher. As the overall order is held constant, the same number of files are identified regardless of the orders of the first and second indexes. The time required to identify the files is minimized when cells within the first index become saturated with collisions. For secondary indexes of order 2 or lower, the large increase in performance offered for increases in the index order results from the reduction in the total complexity of the index which translates to shorter times for bitwise operations and less memory

required for storage. Above 2nd order, the overhead from storing and accessing more complex EWAH compressed bitmaps for each collision begins to flatten the memory scaling and increase the time required for queries. However, selections using higher order secondary indices still require less time than in the case where only a single index is used.

The optimal value for the orders of the first and second indexes will depend upon the dataset in question. The density of data points within the test dataset used here is relatively uniform throughout the domain and does not need a high level of refinement at collisions. However, if a datasetwere less uniform with concentrations of points, the optimal order of the second index for performance may be higher.

Selector Size

The time required to identify files touched by a selection will also depend upon the size of the region being selected. Larger selectors will intersect more indices and more files, resulting in more bitmap operations. Figure 19 shows the result from varying the selector size. The same test datasetfrom Section 1.10.2.1 was used. A bitmap index with a 4th order primary Morton index and 2nd order secondary Morton index was used in all cases. Each cube selector was placed at the center of the domain and scaled along each dimension to some fraction of the total domain.
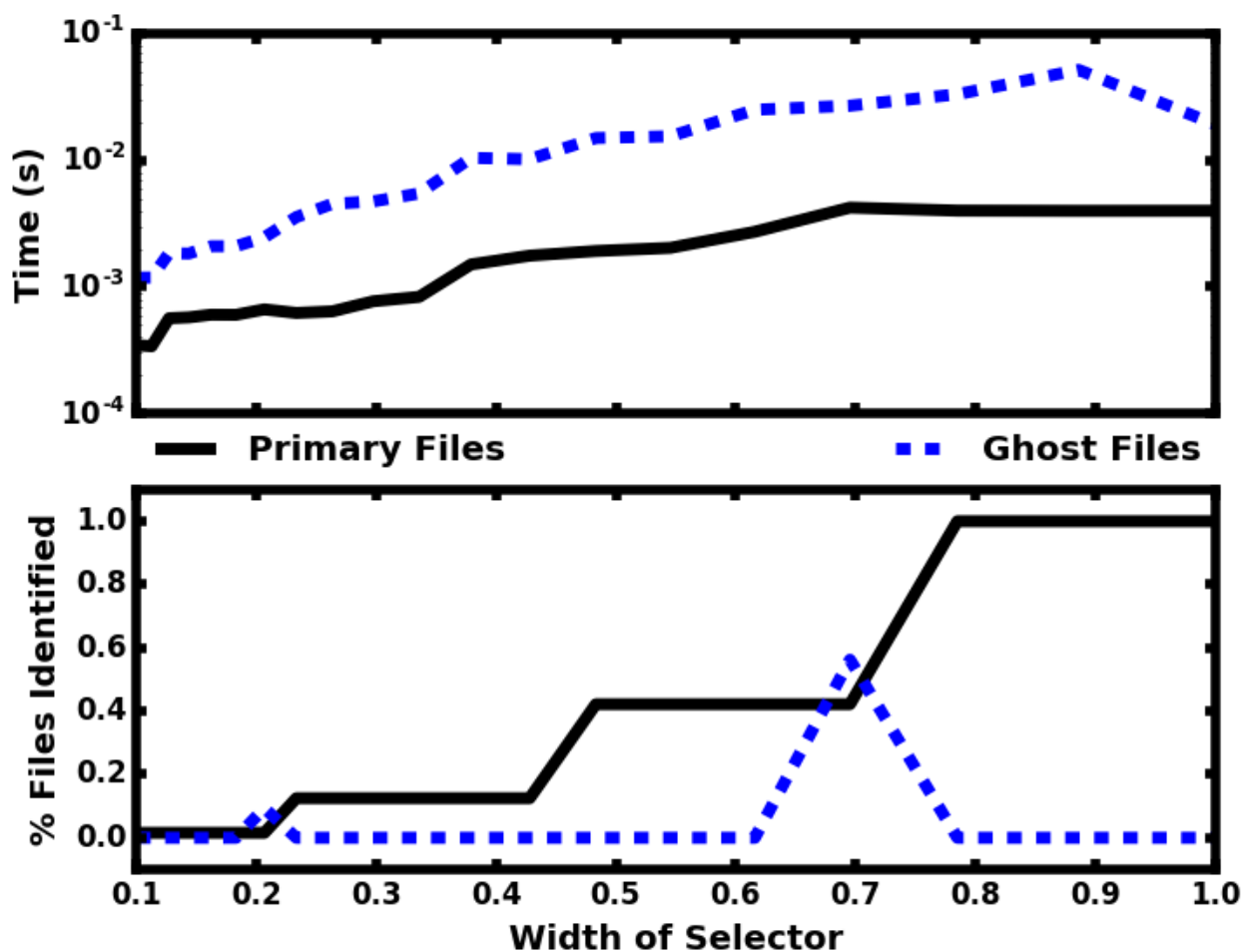
Figure 19: Dependence of query time (top) and number of files selected (bottom) on selector width in terms of the total domain width. The solid black lines correspond to the query times and files identified by the selectors alone. The dashed blue lines correspond the query times and additional files identified when a ghost zone with a width of one cell is added to the selector.

As the selector increases in size, it touches a greater number of files, resulting in longer query times. The number of files touched increases in steps due to the way the test dataset was partitioned between files. Using the Hilbert curve, the domain covered by any one file is localized and will have a rectangular shape. This results in an ordered structure that is similar along all dimensions. An increase in the number of files touched indicates that the selector has grown past a file boundary in all directions. It is just prior to these jumps that ghost files are present. If the selector edge is near a file boundary, ghost zones have the potential to overlap the domains contained by neighboring files that are not already touched by the selector. For such a highly ordered dataset, the ghost zones will only identify additional files for selectors that are nearing the edges of file boundaries. However, queries including ghost zones require slightly more time even when this is not the case.

Domain Partitioning

As discussed in Section ??, a bitmap index is more effective in cases where the domain is partitioned between files in a localized way. If files contain non-contiguous parts of the domain, contiguous selections will require more files to be loaded. Figure 20 shows results for four different partitioning schemes. All four data sets cover the same three-dimensional domain with 1024^3 points split across 512 files. The Hilbert dataset is the same one used in previous tests (see Section 1.10.2.1 for a description). The Morton dataset is constructed in a similar way to the Hilbert dataset with file partitions occurring along a 6th order Morton curve and including a 10% scatter of points between Morton cells. The sliced dataset is partitioned in slices along one dimension with 10% scatter of points between adjacent slices. Files in the random dataset contain a random sample of points, uniformly distributed across the domain.
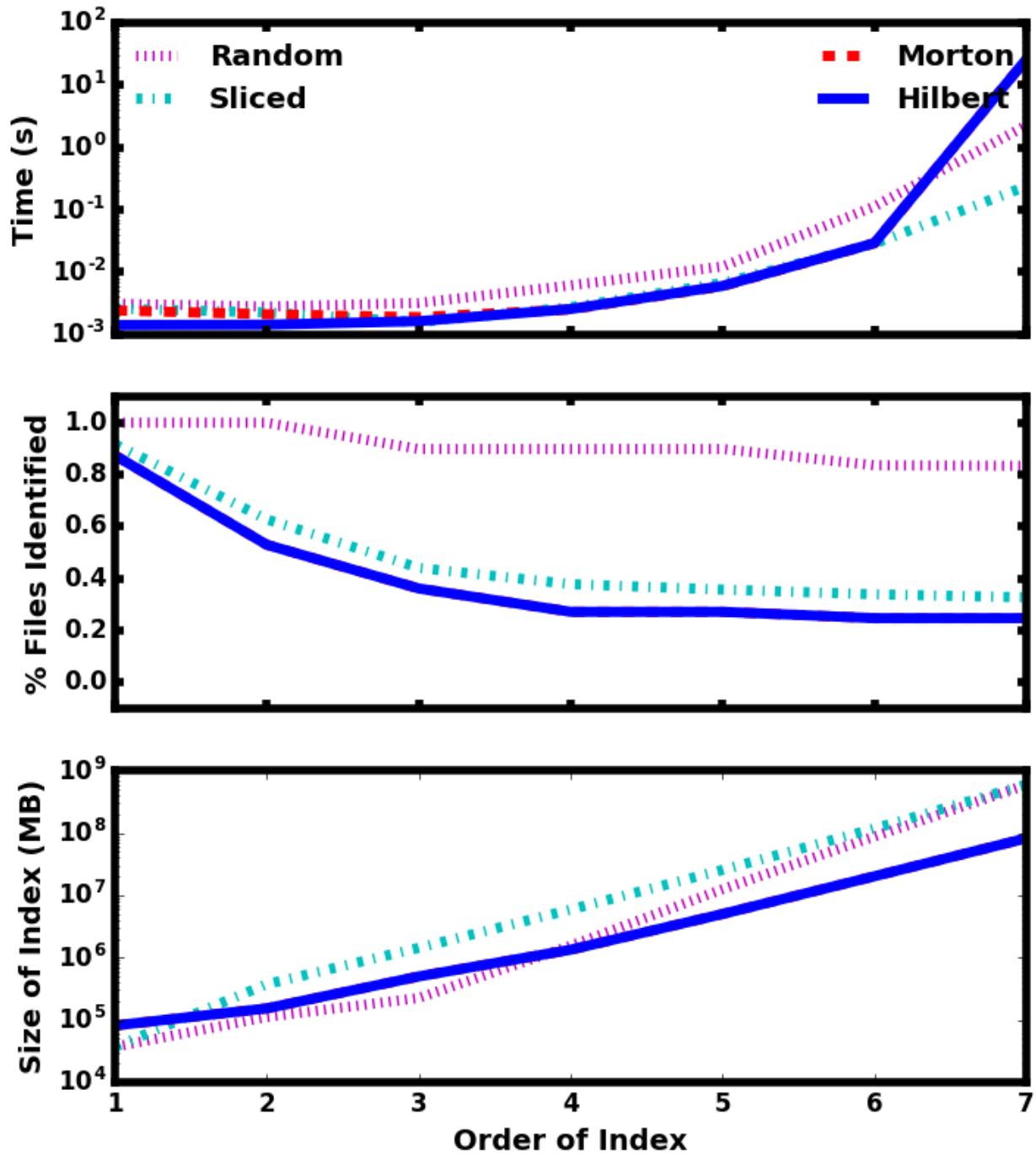
Figure 20: Dependence of query time (top), the number of files selected (middle), and the size of the index (bottom) on index order for different domain partitioning between files. The dotted magenta lines are for a randomly partitioned dataset, the cyan dashed-dotted lines are a dataset partitioned by equal slices alone one dimension, the dashed red lines are a dataset partitioned along an 6th order Morton curve, and the solid blue lines are a dataset partitioned along a 6th order Hilbert curve. **NOTE: it looks like the Morton curve is almost always behind the Hilbert one. Can you use zorder to put it at the frontrow instead so that we can see both ?**

Many more files are identified for the random dataset than those datasets with localized partitioning of the domain. Above an order of 3, very few files could be excluded for the random dataset. This was not true for the localized partitioning schemes. At the highest order, only $\sim 20 - 30\%$ of the files within these datasets would need to be loaded in order to get all of the data within the selected regions, while $\sim 80\%$ of the files in the random dataset would be required. The smallest fraction of files were identified for the Hilbert and Morton datasets, with a slightly greater fraction being identified for the

sliced dataset. For a 6th order index and below, queries on the Morton and Hilbert partitioned datasets are the fastest. An index order of 7 provides refinement beyond the 6th order curves used to partition the dataset between the files and the required for queries on these datasets increases dramatically. The sliced dataset performed particularly well in this case because the selectors used were cubes and did not preferentially select along any one dimension.

Overall, this technique offers a considerable improvement in performance over other methods that require reading, evaluating and discarding all of the particles.

Summary & Discussion

Mapping files using Morton bitmap indexes speeds up analysis of large datasets split across multiple files by reducing the number of files that need to be loaded in order to perform operations on a subset of the full domain. The time required for making selections using the bitmap index is minimal for even large datasets and can be optimized by partitioning the domain between files in a localized way and using an index or indexes of appropriate order for the dataset.

Without an index, queries require loading the data contained in every file into memory and then searching the data for those points that are selected by the query. With a bitmap index, queries require loading the index, using it to identify the files touched by the query, reading in the data contained within the identified files, and searching the data for points selected by the query. In this way, the bitmap index can decrease the computational cost of reading in the data and selecting data points if it identifies a subset of the total number of files. While using an existing bitmap index decreases the time required for queries in this case, constructing a bitmap index can be more computationally expensive than directly querying the data without a bitmap index. Therefore, in the case where only a small number of selections need to be made, it will be more efficient to perform direct queries of the data than to construct and utilize the bitmap index.

Bitmap indexing is particularly useful in astronomy and astrophysics. Output from N-body simulations is often split between multiple files to take advantage of parallel I/O and the domain decomposition generally leads to localized partitioning between files [8,28,29].

Currently, this technique is most useful for datasets split across multiple files. However, it can also be applied to single files by dividing the file's contents into chunks. As in the multi-file case, the single file would need to be organized such that chunks were localized within the domain to take full advantage of the bitmaps. In addition, while the current implementation of this method is designed for three-dimensional spatial datasets like those produced by astrophysical simulations, the same methods can be applied to non-spatial datasets with arbitrary dimensionality.

Code

These procedures have been implemented as part of the yt Python package [30] in order to facilitate the analysis of large astrophysical N-body simulations; currently undergoing review for inclusion in a future version of yt (4.0 or later), our implementation is available at https://bitbucket.org/langmm/yt-bitmap. **NOTE: This use to say "3.4 or later". Now it's pretty clear this section is a bit dated. Hasn't the bitmap stuff been merged by now ?** The open source EWAHBoolArray C++ package is used for implementing EWAH bitmaps [25,26] and exposed to Python using Cython [31].

Acknowledgment

The authors would like to thank Daniel Lemire for his open source EWAH implementation. yt is developed by a large number of independent researchers from numerous institutions around the world. Their commitment to open science has helped make this work possible.

Data Formats and Frontends

adaptahop

Index Type: particle

Comments: Halo finding format

ahf

Index Type: particle

Comments: Halo finding format

amrvac

Index Type: grid

Comments: MPI-AMRVAC is a parallel adaptive mesh refinement framework aimed at solving (primarily hyperbolic) partial differential equations by a number of different numerical schemes. The emphasis is on (near) conservation laws and on shock-dominated problems in particular.

Primary Citations: [32,33,34]

Usage Citations: [35,36,37]

arepo

Index Type: particle

Comments: While listed as a "particle" dataset, AREPO data is properly described as a moving mesh. For visualization purposes, yt treats it similarly to an SPH dataset; some efforts have been made to properly apply nearest-neighbor interpolation, but these have not been mainlined in the yt visualization routines as of yet.

Usage Citations: [38,39]

art

Index Type: octree, particle

Comments: ART, or 'NMSU-ART' as it is occasionally referred to internally, is an octree-based code that is often used for cosmology simulations and galaxy formation simulations. yt supports two sets of data formats in the NMSU-ART family. The first is a particle-only format, where the particles are stored independently of any gas structure. The second is the hydro data format, wherein gas values are stored organized into an octree. Each of these constitutes a different indexing system, sharing similar datatypes, units, and so forth.

Usage Citations: [38,40]

artio

Index Type: octree

Comments: The ARTIO frontend, which shares some ancestry with the `art` or 'NMSU-ART' frontend, utilizes a special purpose library for input/output operations called `libartio`, which is bundled with yt.

Utilizing `libartio` is interesting from a historical perspective, as it provided an opportunity to reorganize the way that yt understood indexing of octree datasets. Whereas previously yt required a full understanding of data distribution, the `artio` frontend outsourced much of that to the internal library, which it regarded as essentially an opaque API. Reducing the amount of internal bookkeeping that yt had to do provided a lighter-weight interface to the data, which in turn has enabled other frontends to be created using opaque APIs.

Usage Citations: [41]

athena

Index Type: grid

Comments: None

Usage Citations: [42]

athena_pp

Index Type: grid

Comments: None

Usage Citations: [43]

boxlib

Index Type: grid

Comments: None

Primary Citations: [44,45,46,47]

Usage Citations: [[48]; https://doi-org.colorado.idm.oclc.org/10.1007/s00162-020-00548-6; https://doi.org/10.1103/PhysRevD.100.063528 , https://doi.org/10.1103/PhysRevD.103.063525 , https://doi.org/10.1103/PhysRevD.105.023516; DOI:10.1016/j.cpc.2022.108457]

chombo

Index Type: grid

Comments: None

Usage Citations: [49,50,51]

eagle

Index Type: particle

Comments: None

Usage Citations: [52,52,53,54]

enzo

Index Type: grid

Comments: Enzo was the very first code supported by yt, and while many remnants of that have been removed, many still remain. During the development of yt 3.0 in particular, many "enzo-isms" were removed; however, as Enzo uses a patch-based grid data format, much of the naming conventions for grid objects still mirror those in use in Enzo. These include names such as `GridLeftEdge` and `ActiveDimensions` (which refers to the dimensions not including ghost zones).

There are several 'sub-frontends' within the Enzo frontend; these include dedicated 1D and 2D readers, as well as historical frontends (of varying functionality) for reading HDF4, 'unpacked' HDF5 and other sub-dialects of Enzo.

Primary Citations: [55]

Usage Citations: [38,56,57,58,59,60]

enzo_e

Index Type: grid

Comments: Enzo-E is a nearly full-rewrite of the Enzo infrastructure designed for emerging and future architectures.

Usage Citations: [61]

exodus_ii

Index Type: unstructured mesh

Comments: Exodus II, and its output in NetCDF4 format, was the initial driver for higher-order unstructured mesh support in yt. The original driver for developing this format was for simulations of nuclear reactors.

fits

Index Type: grid

Comments: None

flash

Index Type: grid

Comments: FLASH is a ...

Usage Citations: [62,63,64,65,66,67]

gadget

Index Type: particle

Comments: None

Usage Citations: [68]

gadget_fof

Index Type: particle

Comments: Halo finding

gamer

Index Type: grid

Comments: None

Primary Citations: [69]

gdf

Index Type: grid

Comments: Custom built for yt

gizmo

Index Type: particle

Comments: None

Usage Citations: [38,70]

halo_catalog

Index Type: particle

Comments: None

http_stream

Index Type: particle

Comments: None

moab

Index Type: unstructured mesh

Comments: None

nc4_cm1

Index Type: grid

Comments: None

open_pmd

Index Type: particle

Comments: None

owls

Index Type: particle

Comments: None

owls_subfind

Index Type: particle

Comments: None

ramses

Index Type: octree

Comments: None

Usage Citations: [38,71,72,73,74,75,76]

rockstar

Index Type: particle

Comments: None

sdf

Index Type: particle

Comments: None

Usage Citations: [77]

sph

Index Type: particle

Comments: A base frontend from which other SPH-type frontends can be defined.

stream

Index Type: multi

Comments: None

Usage Citations: [78,79]

swift

Index Type: particle

Comments: None

Primary Citations: [80]

Usage Citations: [[81]; https://ui.adsabs.harvard.edu/abs/2022arXiv220313915B]

tipsy

Index Type: particle

Comments: None

Usage Citations: [82,83]

ytdata

Index Type: particle

Comments: An internal frontend for re-loading saved data and derived data

Usage Citations: [84]

The primary method by which researchers interact with their data in `yt` is via visualization; from the standpoint of the library, however, this is a side-effect of the various analysis, regularization and data-processing algorithms that are implemented within `yt`. Nearly all of the visualization that is done using `yt` utilizes the matplotlib library for actual deposition of pixels into an image format, although all of the *input* to that deposition is conducted by `yt`. Making this distinction is important, because it

underscores the relationship between the different libraries and how they exist in the ecosystem of scientific software; `yt` does not replace matplotlib, but rather, augments it by providing a grammar of analysis of volumetric data and defining how that grammar is translated into visual representations as presented by matplotlib.

CC: discuss ray traversal for patch-based datasets + oct-based datasets.

Pixelizing Variable-Mesh Objects

The results of either projecting or slicing through a logically-cartesian finite volume dataset is represented in `yt` as a collection of pixel positions and widths. These objects, hereafter referred to as exposing the "variable mesh" interface (as originated in HippoDraw), are not typically suitable for direct visualization. Many visualization libraries, including matplotlib, would necessarily regard these as collections of patches of fixed size, supplying them to the underlying engine. To optimize for repeated rendering, `yt` provides its own "pixelization" routines that take advantage of the input data structures. These "pixelizers" (or "rasterizers") can account for periodic data, variable resolution, overlapping and disjoint datasets, and non-Cartesian coordinate systems.

The pixelizers in `yt` are implemented in Cython, and they accept an input "image plane" buffer (with extent) as well as the variable mesh to be deposited. Pixelizers exist for cartesian coordinates, cylindrical and spherical coordinates, off-axis cartesian planes, and for the Mollweide orthographic projection. Each of these pixelizers follows a roughly identical process for depositing source pixels into the image plane. The outer loop is over the input pixels, p_i, composed of $x_i, y_i, dx_i, dy_i, v_i$, where x and y refer to the coordinate system; in practice this means they may actually represent the r, θ, ϕ or other coordinates.

1. Compute left and right edges of the bounding box for this pixel in the resolution of the image plane
2. Iterate over the first image plane coordinate from the left edge to the right edge of the bounding box
3. Iterate over the second image plane coordinate from the left edge to the right edge of the bounding box
4. Map from the coordinate system to the image plane and deposit v

In practice, this is a fast operation, as long as the inner loops are sufficiently well determined; for instance, when depositing an input pixel with a width of w into an image plane where the pixel width corresponds to a width of $w/16$, only 16^2 pixels (with a high-degree of sequential ordering) have to be iterated over. The spherical and cylindrical pixelization routines operate similarly, but are somewhat degraded by a lower degree of locality in the final mapping from coordinate system to image plane.

Recent work has been done to port the pixelization routines to Rust and compiling these to WebAssembly, resulting in the development of the Widgyts project. Widgyts provides a browser-side Jupyterlab interface to the pixelization routines, enabling extremely low-latency exploration of datasets.

Higher-Order Unstructured Mesh Elements

Software Volume Rendering

The volume rendering is based on classical concepts for rendering 3D objects, and relies on the notion of a scene, a camera and an object to render. The object to render can be any data container of supported AMR datasets (either patch-based and octree-based datasets). The implementation of the volume rendering is based on integrating a transfer function along individual "rays." By defining a traversal function, cells can be ordered and a sampling function and accumulation can be applied to

them. Below, we describe the traversal operations, and then the different "sampling" operations that can be applied.

Conceptually, it is easies to think about this as constructing a global "ordering" of the cells, and then an in-order application of the sampler function with affiliated ray-specific data storage. In reality, however, these operations are interleaved and decomposed to enable task- and data-based parallelism. Data is distributed to different processors wherever possible based on different decomposition strategies, and within the traversal algorithms OpenMP parallelism is utilized to allow for individual rays to be integrated simultaneously. Typically, multiple "sampling points" are identified within each element to be sampled; the default number is 5. We have found that the image quality improves somewhat as this number is increased, as the interpolation allows for finer-grained variation, but 5 is typically sufficient for all but the highest resolution data and highest-gradient data.

Lenses for Volume Rendering

In addition to specifying traversal algorithms, the software volume rendering in yt relies on the concept of "lenses." A lens, in yt terminology, is a descriptive object from which emanate the rays that are integrated. The traditional lens is that of the perspective projection, wherein a plane of rays are sent outward. The simplest lens, however, is that of a plane-parallel set of rays, each of which is emitted by an individual pixel with constant direction. Additional lenses yt provides include a stereo projection lens (for stereoscopic 3D volume rendering), a fisheye lens (for hemispheric rendering suitable for planetarium domes), a spherical lens (for spherical rendering suitable for full-sphere projections (such as for "$360°$ YouTube videos), and the stereoscopic spherical lens.

Each of these lens objects provides a functional form of ray direction and origin. In some cases, an inverse function can be provided, which in some cases enables the traversal function to more intelligently check intersection and reduce computation time.

Patch-based ray traversal

Unlike taking standard "projections" of datasets, the process of volume rendering requires that elements be traversed in order, accumulating values that may be reliant on *previous* values. While this is straightforward to implement for grids [85], in situations where there are overlapping sets of values the process must be handled more carefully.
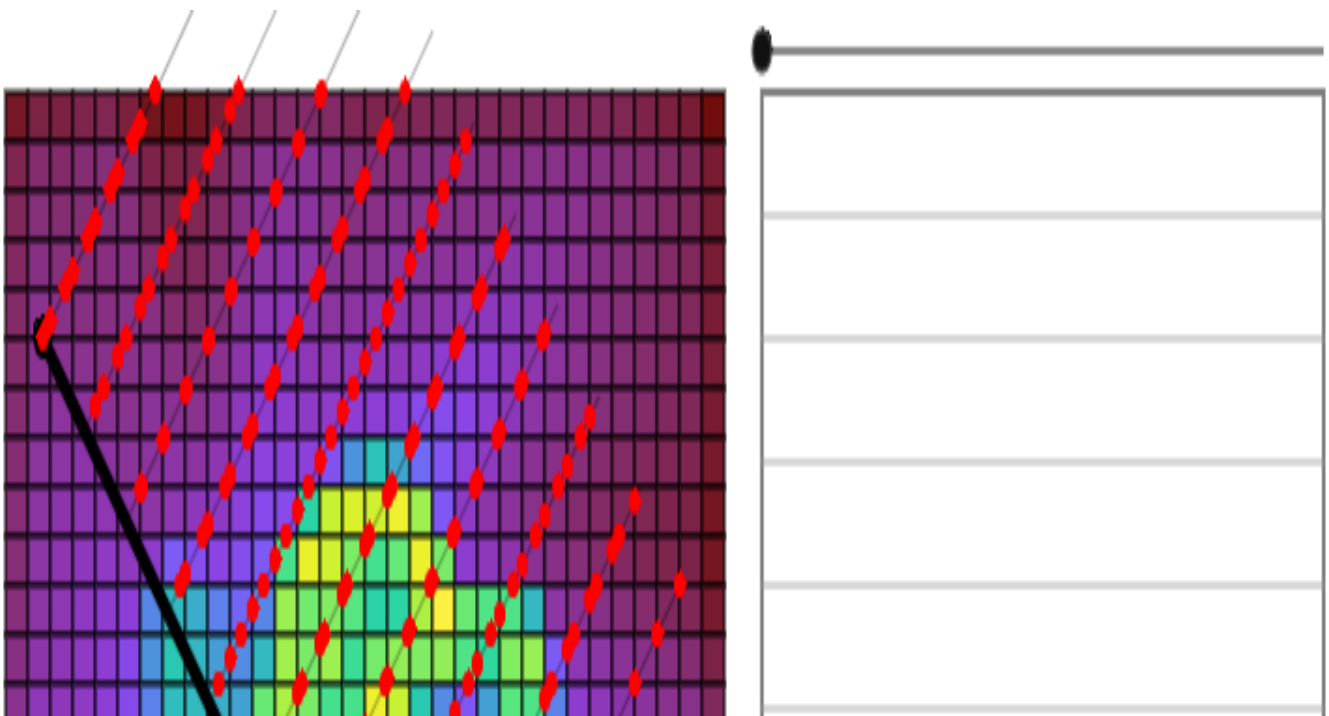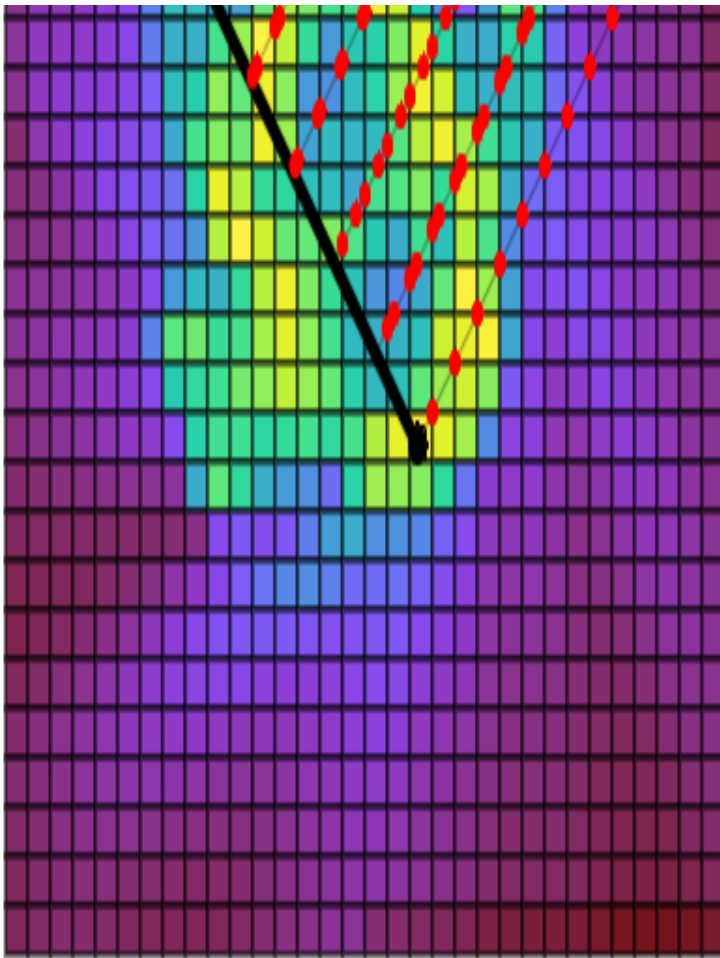
To ensure that cells are traversed in-order, and to ensure that only the highest-resolution cells are used in cases where grids overlap, we decompose the grid hierarchy into a kD-tree. To construct the kD-tree, each grid at the root level is added to the tree. Wherever these grids include "child" grids (i.e., grids that include higher-resolution data that overlaps with the grids) a cutting plane is inserted in the kD-tree such that the child grid is isolated. This process is then applied recursively until all of the grids in the dataset (or its data subselection) have been added. By inserting cutting planes at child-grid boundaries, the tree can be specified to include only "leaf" node data, thus ensuring that any traversal will *only* include the highest-resolution data available, and thus can be traversed in order.

kD-trees also provide so-called "viewpoint traversals," which allows the grids to be ordered in either front-to-back or back-to-front order. These traversals can also be ordered to allow for radially-emanating traversals, for such cases as hemispheric renderings.

Octree ray traversal

Casting rays through an octree can be achieved efficiently by relying on the octree structure. In order to abstract away the underlying layout of the data, we first construct an octree that contains all leaf cells in

the data container. We store all cells as octs with no children, and mark them with their position within the data container, going from 0 to $N_{\text{cell}} - 1$. Octs that are inserted in the process of building the tree are not marked nor indexed. We also compute the vertex-centered data for all cells in the container. Note that, contrary to the octree utilized to index the data from octree datasets, this octree may span multiple domains and contains all levels from the root level (that contains a single oct the size of the simulation domain) all the way down to the leaf cells. We then cast rays off the camera, and for each ray, we compute the ordered list of the N cells it intersects with together with the intersection points along the ray [86]. In the following, we will write the coordinate along the ray as t, with the camera located at $t = 0$. In general the tree may contain holes (this may happen if the data container is a region selector), so that the exit coordinate out of a cell may not coincide with the entry coordinate through the next cell. In practice, we solve this by storing for each cell both the entry and exit coordinates of the ray.

The algorithm relies on the fact that if a ray passes through an oct and intersects with its six faces at coordinate along the ray t_{xi}, t_{yi}, t_{zi} (on entry) and t_{xo}, t_{yo}, t_{zo} (on exit), then its intersection with the inner cells' faces can be computed explicitly from these six values and their half point $(t_{xi} + t_{xo})/2, (t_{yi} + t_{yo})/2, (t_{zi} + t_{zo})/2$. This implies that each call to the algorithm only need computing one division and a few simple arithmetic comparisons. It also uses the fact that for a given oct, we can compute which cell the ray will intersect with first, and from any given cell, which cell the the ray will intersect next.

The algorithm then works as follows. If the ray does not intersect with the root oct, then the algorithm returns an empty list of cell crossed and t values. Otherwise, initialize an empty list of cells traversed and t-values. *a)* Find the intersection of the ray with all six faces of the oct. *b)* If the current oct is marked, store the entry and exit t-values and the index of the oct in their respective list and return. *c)* If the current oct is not marked, find the first cell the ray intersects with and call the algorithm recursively (starting at step *a)* with the oct contained in the cell, if any). *d)* Find the next cell within the oct. If there is no next cell, return. Otherwise, call the algorithm recursively (starting at step *a)* with the oct contained in the cell, if any) then go back to *d)*. On exit of the algorithm, we then have a list of cells and t-values. For each cell in the list, we then call the sampler with the vertex-centered values and the entry and exit coordinates.

An example of the volume rendering of a galaxy in a zoom-in cosmological simulation made with RAMSES is shown on Figure 21.
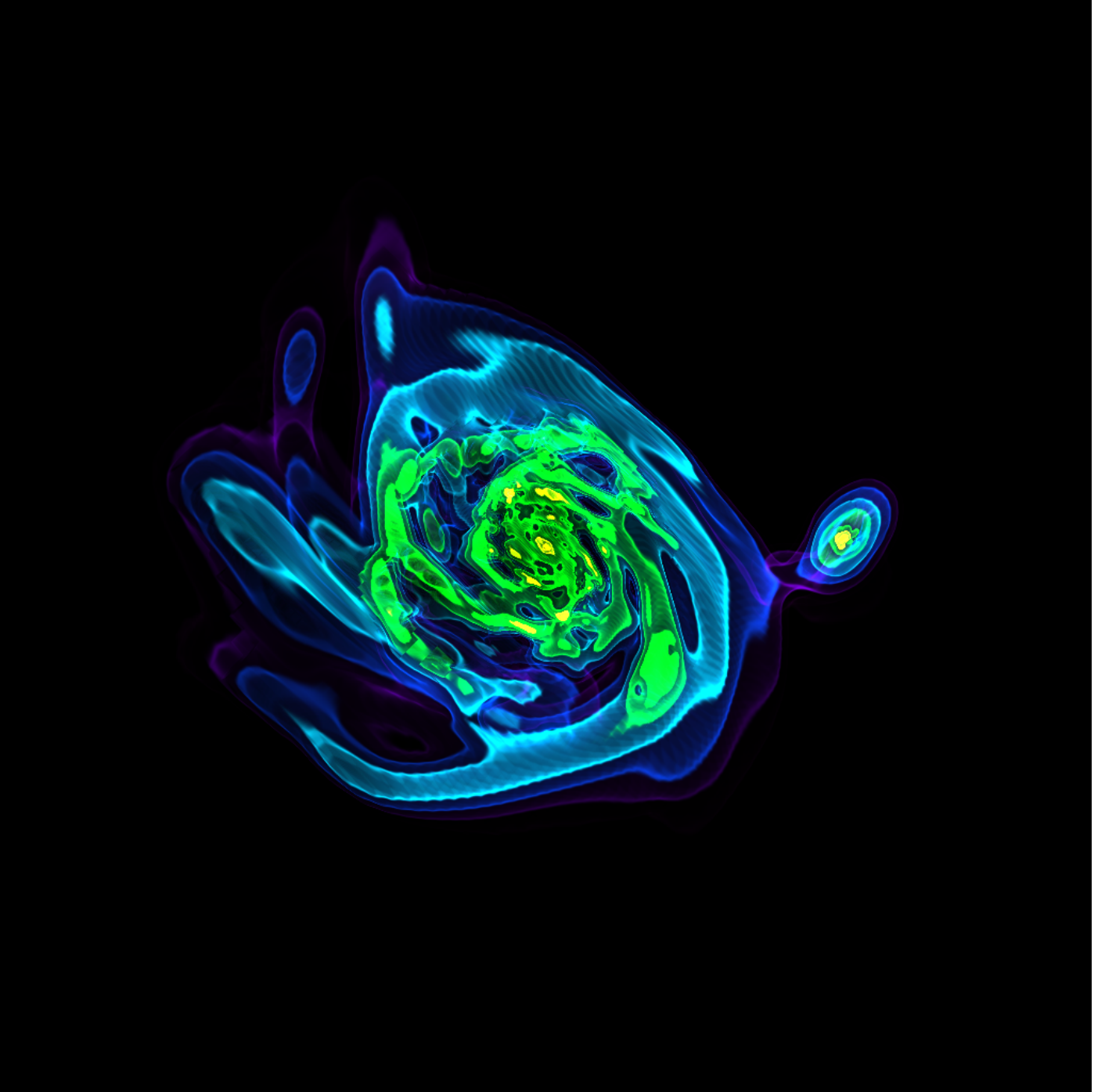
Figure 21: Volume rendering of gas density isocontours around a galaxy in a cosmological zoom-in simulation performed with RAMSES. Adapted from [87].

Sampling functions

As a ray passes through each sampling point, we apply a sampling function. In the yt software volume renderer, these cython subclasses of the `ImageSampler` base class. Each defines a setup process and a sample process. Once a ray arrives at a sampling point (for instance, as it walks across a grid and encounters values) the sampler is called with the data it is accessing, information about the ray, the "enter" and "exit" parameters of the sample region, and the accumulated data so far.

Each `ImageSampler` object implements a different method of integration through the volume. Below, we discuss each in turn. To do so, we apply the convention that our volume rendering ray-casting starts at a position defined by the parameter t, starting at the position $\vec{\mathbf{x}}(t_0) \equiv \vec{\mathbf{x}}_0$ and terminates at $\vec{\mathbf{x}}(t_1) \equiv \vec{\mathbf{x}}_1$. Field values are taken as $f(\vec{\mathbf{x}})$, and the distance through with a ray has passed is defined as the path length.

Projection Sampler

This sampler object conducts a simple integration through the domain.

$$v(\vec{\mathbf{x}}(t_1)) = \int_{t_0}^{t_1} f(x, y, z) dl$$

Because the sampling function includes no dependence on previous values, it can be sampled completely independently and out of order. This allows for much simpler data-based parallelism. There are two formulations of this sampler in yt; the first is a piecewise-linear interpolation scheme, wherein the field is assumed to be constant within each sampling volume. This method requires neither vertex-centered data *nor* multiple sampling points and is considerably faster to conduct. The second method available, which is the default, is that of an interpolated projection. This sampler requires vertex-centered data and utilizes multiple samples. Because the arithmetic is reasonably simple and the data can be supplied in any order, it is still reasonably fast to conduct.

This sampler function operates independently of color channels, exclusively in floating-point space. This allows colormapping to be conducted at the end of the process and allows for high dynamic range beyond that typically enabled by RGBA channels.

Transfer Function Sampler

The transfer function sampler is the most complex of the sampling functions available in yt, as it affords the most flexibility. Unfortunately, this also results in a cumbersome and difficult-to-describe setup process, wherein the simple operations are straightforward to accomplish but the more complex operations are subject to trial-and-error.

The transfer function sampler is designed for constructing images; as such, it provides integration for up to six channels of data. Typically, this is used in the four-channel mode, corresponding to red, green, blue and alpha (RGBA) data. (It is important to note, however, that this is not by any means the only mechanism that these could be assigned; the values are accumulated in floating point and could correspond to any output data, not just these three colors and one alpha.) In six-channel mode, typically the channels are for red, green, blue, and a corresponding alpha channel for each of those colors, accumulated independently.

The transfer function is set up around the concept of "field interpolation tables." These are 1D arrays, typically 256 elements, that include floating point values as well as concrete bounds in data-space. These tables provide a mapping $f(v) \to u$, where v is a local field value, as well as an optional "weight" value drawn from a second table. Careful construction of these tables can produce volume renderings via isocontours, where specific values in field space are highlighted with distinct colors. Indeed, this is the most common method of applying this sampler, as it allows a straightforward way of seeing "through" the outer layers and viewing nesting structures simultaneously.

We use notation C_i^0 for the input values for a given channel (where i is one of R, G, B, A) and C_i^1 for the output. With the sampled field value ($f(t)$) and optional weight field value ($w(t)$) we can then define a sample, S_i for each channel including alpha. we compute the updated values for each sampling point via:

$$C_i^0 a_t \leftarrow \max(1 - dt * S_A, 0)$$
$$C_i^1 \leftarrow S_i \delta t + a_t C_i^0$$

This is the case for "grey" opacity. In the use case of differing opacity for each channel, we modify this such that we utilize additional alpha channels for each primary channel. a_t is then defined individually

for each output channel.

This particular set of transfer functions, with what amounts to multi-channel, multi-field sampling in both output and opacity, can be specified to perform complex functions such as variable scattering, different weights for different channels, etc.

Hardware-accelerated 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 $\mathrm{g/cm^3}$:

```
>>> 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 propagated 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 receive updates if the original unit registry is modified.

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

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

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

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 implementation, 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

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

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

Jupyter Integration

Halo-Finding and Catalogs

Scaling and Parallelism

Performance of Operations

Inline Analysis

Simple Parallelism

Analysis Modules

For much of its development history, `yt` took the approach of bundling as many analysis modules as possible in the primary repository. This provided the advantage of having all work be centralized, and ensuring that each download or installation of `yt` was a fully-featured system for analyzing a large swath of data, but it brought with it the development overhead of the entire `yt` package for what in many cases were isolated pieces of functionality with separable responsibilities.

As a result of the slowing in speed of development as a result of review requirements (and limited personnel to conduct those reviews), some of the analysis modules that were bundled with `yt` have been "spun out" into their own repository, `yt_astro_analysis`. This repository, which is developed, released and installed separately from `yt`, includes modules for cosmological observation (upon which Triden, which is discussed in 1.17.1, is based), dark matter halo finding and analysis, tools for interacting with position-position-velocity cubes, and a system for exporting from `yt` to RADMC-3D. **TODO: Find the correct citation for RADMC-3D.**

Ecosystem of Packages

Several packages have been developed that utilize `yt` as an infrastructural component. In contrast to the analysis modules, which have typically been built as extensions of `yt`, these extensions utilize `yt` as a framework and typically only rely upon public APIs, often with no changes necessary to upstream `yt`. In some cases, `yt` is also an *optional* dependency for the package. In most cases, these are described in detail elsewhere, and citations to that external information are included here.

Trident

Trident [88] is a Python-based open-source tool for post-processing hydrodynamical simulations to produce synthetic absorption spectra and related data. In many ways, Trident is the first external package that utilizes `yt` to provide data access and numerical operations, but then builds on those to develop detailed, astrophysically-aware systems for processing and analyzing that data.

Powderday

Powderday [89,90] is an open source package connecting `yt`, population synthesis modules and the monte carlo radiation transport code Hyperion [91] to develop synthetic observations of galaxies from astrophysics simulations. Powderday uses `yt` to read and transform data, including high-fidelity gridding of SPH particles into an octree form, to prepare galaxy simulations to be processed. Currently Powderday uses the Arepo, Enzo, Gadget, Gizmo, and Tipsy front ends to read in simulation results, and convert them to a common format in preparation for dust radiative transfer (using `unyt` to ensure common units). Plans are in place to employ the wide range of front ends available in yt to broaden the applicability of powderday to a wider range of galaxy simulations.

ytree

Building on `yt` for access to halo catalogs, and implementing a similar system for derived fields as applied to graph datasets, ytree [92] is a system for analyzing merger trees from analysis of dark matter halos in cosmological simulations.

ytree provides flexibility in determining the path that a given analysis takes through the graph of merger trees; for instance, it enables the user to select if they wish to follow the "most massive" progenitor halo backwards in time, or even to set their own criteria for this. Connecting this to the raw, unprocessed data from the simulation (such as the unsampled particle or cell content that comprises the halos) allows researchers to deepen and guide their analysis based on the physical characteristics of the merger history.

pyxsim

pyXSIM [93] is a python package for simulating X-ray observations from astrophysical sources. It implements the PHOX algorithm [94,95] and is implemented to support grid-based codes, particle-based codes, and even data constructed from scratch in numpy arrays. pyXSIM has been used to create synthetic observations from current and upcoming space telescopes.

ytini

While `yt` has functionality for volume rendering, it provides far from the scene-control, composition and camera positioning necessary for "cinematic visualization." The tool ytini [96] serves as an intermediary between `yt` and the visualization platform Houdini, often used for special effects and

visual arts in industry and professional production environments. ytini was developed to ease the process of converting data from simulation format into one suitable for production teams to utilize in Houdini and other visual effects software, and has been designed to enable deeper collaboration between researchers and visualization and outreach experts.

Future Directions

Sustainability

References

1. **Python utilities for Manubot: Manuscripts, open and automated**
   Manubot
   (2023-01-07) https://github.com/manubot/manubot

2. **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/ft6md2
   DOI: 10.1088/0067-0049/192/1/9

3. **widgyts: Custom Jupyter Widgets for Interactive Data Exploration with yt**
   Madicken Munk, Matthew Turk
   Journal of Open Source Software (2020-01-29) https://doi.org/gqbxbf
   DOI: 10.21105/joss.01774

4. **Petascale Cosmology: Simulations of Structure Formation**
   Rupert Croft, Tiziana Di Matteo, Nishikanta Khandai
   Comput. Sci. Eng. (2015-03) http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?
   arnumber=7006381
   DOI: 10.1109/mcse.2015.5

5. **The LSST Data Management System**
   Mario Jurić, Jeffrey Kantor, K-T Lim, Robert H Lupton, Gregory Dubois-Felsmann, Tim Jenness, Tim
   S Axelrod, Jovan Aleksić, Roberta A Allsman, Yusra AlSayyad, … LSST Project
   eprint arXiv:1512.07914 (2015) http://adsabs.harvard.edu/abs/2015arXiv151207914J

6. **A computer oriented geodetic data base and a new technique in file
   sequencing**
   GM Morton
   IBM Ltd. (1996)

7. **Gesammelte Abhandlungen: Band III: Analysis - Grundlagen der
   Mathematik Physik - Verschiedenes Lebensgeschichte**
   David Hilbert
   Springer Berlin Heidelberg (1970) http://dx.doi.org/10.1007/978-3-662-25726-5{_}1
   DOI: 10.1007/978-3-662-25726-5_1 · ISBN: 978-3-662-25726-5

8. **The cosmological simulation code gadget-2**
   Volker Springel
   Mon. Not. R. Astron. Soc. (2005-12) http://adsabs.harvard.edu/abs/2005MNRAS.364.1105S
   DOI: 10.1111/j.1365-2966.2005.09655.x

9. **Cosmological Hydrodynamics with Adaptive Mesh Refinement: a
   new high resolution code called RAMSES**
   Romain Teyssier
   Astron. Astrophys. v.385, p.337-364 (2001-11) http://arxiv.org/abs/astro-ph/0111367
   http://dx.doi.org/10.1051/0004-6361:20011817
   DOI: 10.1051/0004-6361:20011817

10. **Speeding up construction of PMR quadtree-based spatial
    indexes**
    Gisli R Hjaltason, Hanan Samet

VLDB J. Int. J. Very Large Data Bases (2002-10) http://link.springer.com/10.1007/s00778-002-0067-8
DOI: 10.1007/s00778-002-0067-8

11. **Fast construction of k-nearest neighbor graphs for point clouds.**
Michael Connor, Piyush Kumar
IEEE Trans. Vis. Comput. Graph. (2010-01) http://www.ncbi.nlm.nih.gov/pubmed/20467058
DOI: 10.1109/tvcg.2010.9 · PMID: 20467058

12. **A class of data structures for associative searching**
JA Orenstein, TH Merrett
Proc. 3rd acm sigact-sigmod symp. Princ. Database syst. - pod. '84 (1984-04)
http://dl.acm.org/citation.cfm?id=588011.588037
DOI: 10.1145/588011.588037 · ISBN: 0897911288

13. **Encoded bitmap indexing for data warehouses**
Ming-Chuan Wu, AP Buchmann
Proc. 14th int. Conf. Data eng. (1998) http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=655780
DOI: 10.1109/icde.1998.655780 · ISBN: 0-8186-8289-2

14. **Bitmap index design and evaluation**
Chee-Yong Chan, Yannis E Ioannidis
ACM SIGMOD Rec. (1998-06) http://dl.acm.org/citation.cfm?id=276305.276336
DOI: 10.1145/276305.276336 · ISBN: 0-89791-995-5

15. **An efficient bitmap encoding scheme for selection queries**
Chee-Yong Chan, Yannis E Ioannidis
ACM SIGMOD Rec. (1999-06) http://dl.acm.org/citation.cfm?id=304181.304201
DOI: 10.1145/304181.304201 · ISBN: 1-58113-084-8

16. **Evaluating Geospatial Geometry and Proximity Queries Using Distributed Hash Tables**
Matthew Malensek, Sangmi Pallickara, Shrideep Pallickara
Comput. Sci. Eng. (2014-07) http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6785924
DOI: 10.1109/mcse.2014.48

17. **Bitmap indexes for large scientific data sets: a case study**
RR Sinha, S Mitra, M Winslett
Proc. 20th ieee int. Parallel distrib. Process. Symp. (2006)
http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1639304
DOI: 10.1109/ipdps.2006.1639304 · ISBN: 1-4244-0054-6

18. **Multi-resolution bitmap indexes for scientific data**
Rishi Rakesh Sinha, Marianne Winslett
ACM Trans. Database Syst. (2007-08) http://portal.acm.org/citation.cfm?doid=1272743.1272746
DOI: 10.1145/1272743.1272746

19. **Improving the Performance of High-Energy Physics Analysis through Bitmap Indices**
Kurt Stockinger, Dirk Duellmann, Wolfgang Hoschek, Erich Schikuta
Database expert syst. Appl. 11th int. Conf. DEXA 2000 london, uk, sept. 4–8, 2000 proc. (2000-06)
http://link.springer.com/10.1007/3-540-44469-6
DOI: 10.1007/3-540-44469-6 · ISBN: 978-3-540-67978-3

20. **Using bitmap index for interactive exploration of large datasets**
Kesheng Wu, W Koegler, J Chen, A Shoshani
15th int. Conf. Sci. Stat. Database manag. 2003. (2003)
http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1214955
DOI: 10.1109/ssdm.2003.1214955 · ISBN: 0-7695-1964-4

21. **Range-based bitmap indexing for high cardinality attributes with skew**
PS Yu
Proceedings. Twenty-second annu. Int. Comput. Softw. Appl. Conf. (Compsac '98) (cat. No.98CB 36241) (1998) http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=716637
DOI: 10.1109/cmpsac.1998.716637 · ISBN: 0-8186-8585-9

22. **Multidimensional indexing and query coordination for tertiary storage management**
A Shoshani, LM Bernardo, H Nordberg, D Rotem, A Sim
Proceedings. Elev. Int. Conf. Sci. Stat. Database manag. (1999)
http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=787637
DOI: 10.1109/ssdm.1999.787637 · ISBN: 0-7695-0046-3

23. **Evaluation Strategies for Bitmap Indices with Binning**
Kurt Stockinger, Kesheng Wu, Arie Shoshani
Database expert syst. Appl. 15th int. Conf. DEXA 2004, zaragoza, spain, august 30-september 3, 2004. Proc. (2004) http://dx.doi.org/10.1007/978-3-540-30075-5{_}12
DOI: 10.1007/978-3-540-30075-5_12 · ISBN: 978-3-540-30075-5

24. **Compressed bitmap indices for efficient query processing**
Kesheng Wu, Ekow Otoo, Arie Shoshani
Lawrence Berkeley Natl. Lab. (2001-09) http://escholarship.org/uc/item/8k22w7q2

25. **Sorting improves word-aligned bitmap indexes**
Daniel Lemire, Owen Kaser, Kamel Aouiche
Data Knowl. Eng. (2010-01) http://dl.acm.org/citation.cfm?id=1663645.1663682
DOI: 10.1016/j.datak.2009.08.006

26. **Compressed bitmap indexes: beyond unions and intersections**
Owen Kaser, Daniel Lemire
Softw. Pract. Exp. (2016-02) http://doi.wiley.com/10.1002/spe.2289
DOI: 10.1002/spe.2289

27. **Smoothed Particle Hydrodynamics**
JJ Monaghan
Annu. Rev. Astron. Astrophys. (1992-09) http://adsabs.harvard.edu/abs/1992ARA{\&}A..30..543M
DOI: 10.1146/annurev.aa.30.090192.002551

28. **GADGET: a code for collisionless and gasdynamical cosmological simulations**
Volker Springel, Naoki Yoshida, Simon DM White
New Astron. (2001-04) http://adsabs.harvard.edu/abs/2001NewA....6...79S
DOI: 10.1016/s1384-1076(01)00042-2

29. **A new class of accurate, mesh-free hydrodynamic simulation methods**
PF Hopkins

Mon. Not. R. Astron. Soc. (2015-04) http://adsabs.harvard.edu/abs/2014arXiv1409.7395H
DOI: 10.1093/mnras/stv195

30. **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
Astrophys. J. Suppl. Ser. (2011-01) http://stacks.iop.org/0067-0049/192/i=1/a=9?key=crossref.ad58f4fc8c6272d93975507bf0238006
DOI: 10.1088/0067-0049/192/1/9

31. **Cython: The Best of Both Worlds**
Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, Kurt Smith
Comput. Sci. Eng. (2011-03) http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5582062
DOI: 10.1109/mcse.2010.118

32. **Adaptive Mesh Refinement for conservative systems: multi-dimensional efficiency evaluation**
R Keppens, M Nool, G Tóth, JP Goedbloed
Computer Physics Communications (2003-07) https://doi.org/ck627b
DOI: 10.1016/s0010-4655(03)00139-5

33. **MPI-AMRVAC FOR SOLAR AND ASTROPHYSICS**
O Porth, C Xia, T Hendrix, SP Moschou, R Keppens
The Astrophysical Journal Supplement Series (2014-08-20) https://doi.org/ggcgbb
DOI: 10.1088/0067-0049/214/1/4

34. **MPI-AMRVAC 2.0 for Solar and Astrophysical Applications**
C Xia, J Teunissen, IEl Mellah, E Chané, R Keppens
The Astrophysical Journal Supplement Series (2018-02-05) https://doi.org/gkh2v6
DOI: 10.3847/1538-4365/aaa6c8

35. **Thermal instabilities: Fragmentation and field misalignment of filament fine structure**
N Claes, R Keppens, C Xia
Astronomy & Astrophysics (2020-04) https://doi.org/gqbk65
DOI: 10.1051/0004-6361/202037616

36. **Prominence formation by levitation-condensation at extreme resolutions**
JM Jenkins, R Keppens
Astronomy & Astrophysics (2021-02) https://doi.org/gqbk66
DOI: 10.1051/0004-6361/202039630

37. **Dynamical signatures of Rossby vortices in cavity-hosting disks**
CMT Robert, H Méheut, F Ménard
Astronomy & Astrophysics (2020-09) https://doi.org/gqbk64
DOI: 10.1051/0004-6361/201937414

38. **High Angular Momentum Halo Gas: A Feedback and Code-independent Prediction of LCDM**
Kyle R Stewart, Ariyeh H Maller, Jose Oñorbe, James S Bullock, MRyan Joung, Julien Devriendt, Daniel Ceverino, Dušan Kereš, Philip F Hopkins, Claude-André Faucher-Giguère
The Astrophysical Journal (2017-06-30) https://doi.org/gf48kq
DOI: 10.3847/1538-4357/aa6dff

39. **Fast galaxy bars continue to challenge standard cosmology**

Mahmood Roshan, Neda Ghafourian, Tahere Kashfi, Indranil Banik, Moritz Haslbauer, Virginia Cuomo, Benoit Famaey, Pavel Kroupa
Monthly Notices of the Royal Astronomical Society (2021-09-11) https://doi.org/gqsz56
DOI: 10.1093/mnras/stab2553

40. **O <scp>vi</scp> traces photoionized streams with collisionally ionized boundaries in cosmological simulations of <i>z</i> ~ 1 massive galaxies**
Clayton Strawn, Santi Roca-Fàbrega, Nir Mandelker, Joel Primack, Jonathan Stern, Daniel Ceverino, Avishai Dekel, Bryan Wang, Rishi Dange
Monthly Notices of the Royal Astronomical Society (2020-12-24) https://doi.org/grk7k2
DOI: 10.1093/mnras/staa3972

41. **Testing feedback from star clusters in simulations of the Milky Way formation**
Gillen Brown, Oleg Y Gnedin
Monthly Notices of the Royal Astronomical Society (2022-04-28) https://doi.org/grk7k3
DOI: 10.1093/mnras/stac1164

42. **Modeling UV Radiation Feedback from Massive Stars. I. Implementation of Adaptive Ray-tracing Method and Tests**
Jeong-Gyu Kim, Woong-Tae Kim, Eve C Ostriker, MAaron Skinner
The Astrophysical Journal (2017-12-15) https://doi.org/gqsz6c
DOI: 10.3847/1538-4357/aa9b80

43. **A Magnetized, Moon-forming Giant Impact**
PD Mullen, CF Gammie
The Astrophysical Journal (2020-10-29) https://doi.org/gqsz6j
DOI: 10.3847/2041-8213/abbffd

44. **AMReX: a framework for block-structured adaptive mesh refinement**
Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, … Michael Zingale
Journal of Open Source Software (2019-05-12) https://doi.org/ghr39d
DOI: 10.21105/joss.01370

45. **Meeting the Challenges of Modeling Astrophysical Thermonuclear Explosions: Castro, Maestro, and the AMReX Astrophysics Suite**
M Zingale, AS Almgren, MG Barrios Sazo, VE Beckner, JB Bell, B Friesen, AM Jacobs, MP Katz, CM Malone, AJ Nonaka, … W Zhang
Journal of Physics: Conference Series (2018-05) https://doi.org/gqsz53
DOI: 10.1088/1742-6596/1031/1/012024

46. **CASTRO: A Massively Parallel Compressible Astrophysics Simulation Code**
Ann Almgren, Maria Sazo, John Bell, Alice Harpole, Max Katz, Jean Sexton, Donald Willcox, Weiqun Zhang, Michael Zingale
Journal of Open Source Software (2020-10-23) https://doi.org/grk7k8
DOI: 10.21105/joss.02513

47. **MAESTROeX: A Massively Parallel Low Mach Number Astrophysical Solver**
Duoming Fan, Andrew Nonaka, Ann Almgren, Donald Willcox, Alice Harpole, Michael Zingale
Journal of Open Source Software (2019-11-21) https://doi.org/grk7k7
DOI: 10.21105/joss.01757

48. **<scp>quokka</scp>: a code for two-moment AMR radiation hydrodynamics on GPUs**
Benjamin D Wibking, Mark R Krumholz
Monthly Notices of the Royal Astronomical Society (2022-02-18) https://doi.org/gpsbw2
DOI: 10.1093/mnras/stac439

49. **The Role of Outflows, Radiation Pressure, and Magnetic Fields in Massive Star Formation**
Anna L Rosen, Mark R Krumholz
The Astronomical Journal (2020-07-23) https://doi.org/grk7k9
DOI: 10.3847/1538-3881/ab9abf

50. **Massive-star Formation via the Collapse of Subvirial and Virialized Turbulent Massive Cores**
Anna L Rosen, Pak Shing Li, Qizhou Zhang, Blakesley Burkhart
The Astrophysical Journal (2019-12-16) https://doi.org/grk7mc
DOI: 10.3847/1538-4357/ab54c6

51. **An unstable truth: how massive stars get their mass**
Anna L Rosen, Mark R Krumholz, Christopher F McKee, Richard I Klein
Monthly Notices of the Royal Astronomical Society (2016-08-25) https://doi.org/f9rwrm
DOI: 10.1093/mnras/stw2153

52. **Revealing the physical properties of gas accreting to haloes in the EAGLE simulations**
Ruby J Wright, Claudia del P Lagos, Chris Power, Camila A Correa
Monthly Notices of the Royal Astronomical Society (2021-04-17) https://doi.org/gqsz55
DOI: 10.1093/mnras/stab1057

53. **Barred spiral galaxies in modified gravity theories**
Mahmood Roshan, Indranil Banik, Neda Ghafourian, Ingo Thies, Benoit Famaey, Elena Asencio, Pavel Kroupa
Monthly Notices of the Royal Astronomical Society (2021-03-08) https://doi.org/gqsz57
DOI: 10.1093/mnras/stab651

54. **Stellar splashback: the edge of the intracluster light**
Alis J Deason, Kyle A Oman, Azadeh Fattahi, Matthieu Schaller, Mathilde Jauzac, Yuanyuan Zhang, Mireia Montes, Yannick M Bahé, Claudio Dalla Vecchia, Scott T Kay, Tilly A Evans
Monthly Notices of the Royal Astronomical Society (2020-11-19) https://doi.org/gqsz54
DOI: 10.1093/mnras/staa3590

55. **ENZO: AN ADAPTIVE MESH REFINEMENT CODE FOR ASTROPHYSICS**
Greg L Bryan, Michael L Norman, Brian W O'Shea, Tom Abel, John H Wise, Matthew J Turk, Daniel R Reynolds, David C Collins, Peng Wang, Samuel W Skillman, …
The Astrophysical Journal Supplement Series (2014-03-20) https://doi.org/gfx7v2
DOI: 10.1088/0067-0049/211/2/19

56. **COLLAPSE AND FRAGMENTATION OF MAGNETIC MOLECULAR CLOUD CORES WITH THE ENZO AMR MHD CODE. I. UNIFORM DENSITY SPHERES**
Alan P Boss, Sandra A Keiser
The Astrophysical Journal (2013-02-01) https://doi.org/gkmgb4
DOI: 10.1088/0004-637x/764/2/136

57. **Predicting Localized Primordial Star Formation with Deep Convolutional Neural Networks**
Azton I Wells, Michael L Norman
The Astrophysical Journal Supplement Series (2021-06-01) https://doi.org/grk7mf
DOI: 10.3847/1538-4365/abfa17

58. **Connecting Primordial Star-forming Regions and Second-generation Star Formation in the Phoenix Simulations**
Azton I Wells, Michael L Norman
The Astrophysical Journal (2022-06-01) https://doi.org/grk7md
DOI: 10.3847/1538-4357/ac6c87

59. **The birth of a galaxy – III. Propelling reionization with the faintest galaxies**
John H Wise, Vasiliy G Demchenko, Martin T Halicek, Michael L Norman, Matthew J Turk, Tom Abel, Britton D Smith
Monthly Notices of the Royal Astronomical Society (2014-06-26) https://doi.org/grk7k6
DOI: 10.1093/mnras/stu979

60. **Evolution of Primordial Magnetic Fields during Large-scale Structure Formation**
Salome Mtchedlidze, Paola Domínguez-Fernández, Xiaolong Du, Axel Brandenburg, Tina Kahniashvili, Shane O'Sullivan, Wolfram Schmidt, Marcus Brüggen
The Astrophysical Journal (2022-04-01) https://doi.org/gp32mc
DOI: 10.3847/1538-4357/ac5960

61. **A Simple Model for Mixing and Cooling in Cloud–Wind Interactions**
Matthew W Abruzzo, Greg L Bryan, Drummond B Fielding
The Astrophysical Journal (2022-02-01) https://doi.org/gqsz6g
DOI: 10.3847/1538-4357/ac3c48

62. **Type Ia supernova ejecta–donor interaction: explosion model comparison**
C McCutcheon, Y Zeng, Z-W Liu, RG Izzard, K-C Pan, H-L Chen, Z Han
Monthly Notices of the Royal Astronomical Society (2022-05-10) https://doi.org/gqsz58
DOI: 10.1093/mnras/stac1275

63. **Exploring Fundamentally Three-dimensional Phenomena in High-fidelity Simulations of Core-collapse Supernovae**
Evan P O'Connor, Sean M Couch
The Astrophysical Journal (2018-09-25) https://doi.org/gjctnx
DOI: 10.3847/1538-4357/aadcf7

64. **Implementing primordial binaries in simulations of star cluster formation with a hybrid MHD and direct *N*-body method**
Claude Cournoyer-Cloutier, Aaron Tran, Sean Lewis, Joshua E Wall, William E Harris, Mordecai-Mark Mac Low, Stephen LW McMillan, Simon Portegies Zwart, Alison Sills
Monthly Notices of the Royal Astronomical Society (2020-12-18) https://doi.org/grk7kz
DOI: 10.1093/mnras/staa3902

65. **Collisional *N*-body Dynamics Coupled to Self-gravitating Magnetohydrodynamics Reveals Dynamical Binary Formation**
Joshua E Wall, Stephen LW McMillan, Mordecai-Mark Mac Low, Ralf S Klessen, Simon Portegies Zwart
The Astrophysical Journal (2019-12-11) https://doi.org/grk7mb
DOI: 10.3847/1538-4357/ab4db1

66. **Modeling of the Effects of Stellar Feedback during Star Cluster Formation Using a Hybrid Gas and N-Body Method**
Joshua E Wall, Mordecai-Mark Mac Low, Stephen LW McMillan, Ralf S Klessen, Simon Portegies Zwart, Andrew Pellegrino
The Astrophysical Journal (2020-12-01) https://doi.org/gphr2s
DOI: 10.3847/1538-4357/abc011

67. **Winds in Star Clusters Drive Kolmogorov Turbulence**
Monica Gallegos-Garcia, Blakesley Burkhart, Anna L Rosen, Jill P Naiman, Enrico Ramirez-Ruiz
The Astrophysical Journal (2020-08-19) https://doi.org/grk7mg
DOI: 10.3847/2041-8213/ababae

68. **The AGORA High-resolution Galaxy Simulations Comparison Project. III. Cosmological Zoom-in Simulation of a Milky Way–mass Halo**

Santi Roca-Fàbrega, Ji-hoon Kim, Loic Hausammann, Kentaro Nagamine, Alessandro Lupi, Johnny W Powell, Ikkoh Shimizu, Daniel Ceverino, Joel R Primack, Thomas R Quinn, … Matthew J Turk
The Astrophysical Journal (2021-08-01) https://doi.org/gqsz6d
DOI: 10.3847/1538-4357/ac088a

69. **gamer-2: a GPU-accelerated adaptive mesh refinement code – accuracy, performance, and scalability**
Hsi-Yu Schive, John A ZuHone, Nathan J Goldbaum, Matthew J Turk, Massimo Gaspari, Chin-Yu Cheng
Monthly Notices of the Royal Astronomical Society (2018-09-24) https://doi.org/gkvtzd
DOI: 10.1093/mnras/sty2586

70. **The galactic dust-up: modelling dust evolution in FIRE**
Caleb R Choban, Dušan Kereš, Philip F Hopkins, Karin M Sandstrom, Christopher C Hayward, Claude-André Faucher-Giguère
Monthly Notices of the Royal Astronomical Society (2022-06-07) https://doi.org/gqsz59
DOI: 10.1093/mnras/stac1542

71. **Stellar angular momentum can be controlled from cosmological initial conditions**
Corentin Cadiou, Andrew Pontzen, Hiranya V Peiris
arXiv (2022) https://doi.org/grk7mh
DOI: 10.48550/arxiv.2206.11913

72. **Accurate tracer particles of baryon dynamics in the adaptive mesh refinement code Ramses**
Corentin Cadiou, Yohan Dubois, Christophe Pichon
Astronomy & Astrophysics (2019-01) https://doi.org/gf42rj
DOI: 10.1051/0004-6361/201834496

73. **Gravitational torques dominate the dynamics of accreted gas at <i>z</i> &gt; 2**
Corentin Cadiou, Yohan Dubois, Christophe Pichon
Monthly Notices of the Royal Astronomical Society (2022-06-17) https://doi.org/grk7k4
DOI: 10.1093/mnras/stac1663

74. **Dense gas formation and destruction in a simulated Perseus-like galaxy cluster with spin-driven black hole feedback**
RS Beckmann, Y Dubois, P Guillard, P Salome, V Olivares, F Polles, C Cadiou, F Combes, S Hamer, MD Lehnert, G Pineau des Forets
Astronomy & Astrophysics (2019-10-21) https://doi.org/grk7kw
DOI: 10.1051/0004-6361/201936188

75. **Gas accretion and ram pressure stripping of haloes in void walls**
BB Thompson, R Smith, K Kraljic
Monthly Notices of the Royal Astronomical Society (2022-10-19) https://doi.org/grk7k5
DOI: 10.1093/mnras/stac2963

76. **The Gaia-ESO Survey: matching chemodynamical simulations to observations of the Milky Way**
BB Thompson, CG Few, M Bergemann, BK Gibson, BA MacFarlane, A Serenelli, G Gilmore, S Randich, A Vallenari, EJ Alfaro, … S Zaggia
Monthly Notices of the Royal Astronomical Society (2017-09-11) https://doi.org/gc8ws7
DOI: 10.1093/mnras/stx2316

77. **Turnaround radius of galaxy clusters in <i>N</i>-body simulations**
Giorgos Korkidis, Vasiliki Pavlidou, Konstantinos Tassis, Evangelia Ntormousi, Theodore N Tomaras, Konstantinos Kovlakas

Astronomy & Astrophysics (2020-07) https://doi.org/gqsz52
DOI: 10.1051/0004-6361/201937337

78. **3D volume rendering of geophysical data using the yt platform**
Christopher Havlin, Benjamin Holtzman, Kacper Kowalik, Madicken Munk, Samantha Walkow, Matthew Turk
Wiley (2021-02-10) https://doi.org/grk7kv
DOI: 10.1002/essoar.10506118.2

79. **Understanding Heating in Active Region Cores through Machine Learning. I. Numerical Modeling and Predicted Observables**
WT Barnes, SJ Bradshaw, NM Viall
The Astrophysical Journal (2019-07-25) https://doi.org/grk7m6
DOI: 10.3847/1538-4357/ab290c · PMID: 31920207 · PMCID: PMC6951256

80. **swiftsimio: A Python library for reading SWIFT data**
Josh Borrow, Alexei Borrisov
Journal of Open Source Software (2020-08-01) https://doi.org/gqsz6b
DOI: 10.21105/joss.02430

81. **sígame v3: Gas Fragmentation in Postprocessing of Cosmological Simulations for More Accurate Infrared Line Emission Modeling**
Karen Pardos Olsen, Blakesley Burkhart, Mordecai-Mark Mac Low, Robin G Treß, Thomas R Greve, David Vizgan, Jay Motka, Josh Borrow, Gergö Popping, Romeel Davé, … Desika Narayanan
The Astrophysical Journal (2021-11-01) https://doi.org/gqsz6f
DOI: 10.3847/1538-4357/ac20d4

82. **AstroBlend: An astrophysical visualization package for Blender**
JP Naiman
Astronomy and Computing (2016-04) https://doi.org/f8phr9
DOI: 10.1016/j.ascom.2016.02.002

83. **The anatomy of a star-forming galaxy: pressure-driven regulation of star formation in simulated galaxies**
SM Benincasa, J Wadsley, HMP Couchman, BW Keller
Monthly Notices of the Royal Astronomical Society (2016-07-20) https://doi.org/f86kmr
DOI: 10.1093/mnras/stw1741

84. **The growth of black holes from Population III remnants in the Renaissance simulations**
Britton D Smith, John A Regan, Turlough P Downes, Michael L Norman, Brian W O'Shea, John H Wise
Monthly Notices of the Royal Astronomical Society (2018-08-02) https://doi.org/gfn3dp
DOI: 10.1093/mnras/sty2103

85. **A Fast Voxel Traversal Algorithm for Ray Tracing**
John Amanatides, Andrew Woo
EG 1987-Technical Papers (1987) https://doi.org/gq2qd9
DOI: 10.2312/egtp.19871000

86. **An Efficient Parametric Algorithm for Octree Traversal**
Jorge Revelles, Carlos Ureña, Miguel Lastra
Journal of WSCG http://wscg.zcu.cz/wscg2000/Papers_2000/X31.pdf

87. **Gravitational torques dominate the dynamics of angular momentum of the accreted gas at $z>2$**
Corentin Cadiou, Yohan Dubois, Christophe Pichon

in prep.

88. **Trident: A Universal Tool for Generating Synthetic Absorption Spectra from Astrophysical Simulations**
Cameron B Hummels, Britton D Smith, Devin W Silvia
The Astrophysical Journal (2017-09-20) https://doi.org/gcx4th
DOI: 10.3847/1538-4357/aa7e2d

89. **powderday: Dust Radiative Transfer for Galaxy Simulations**
Desika Narayanan, Matthew J Turk, Thomas Robitaille, Ashley J Kelly, BConnor McClellan, Ray S Sharma, Prerak Garg, Matthew Abruzzo, Ena Choi, Charlie Conroy, … John H Wise
The Astrophysical Journal Supplement Series (2021-01-13) https://doi.org/gpxcx8
DOI: 10.3847/1538-4365/abc487

90. **The formation of submillimetre-bright galaxies from gas infall over a billion years**
Desika Narayanan, Matthew Turk, Robert Feldmann, Thomas Robitaille, Philip Hopkins, Robert Thompson, Christopher Hayward, David Ball, Claude-André Faucher-Giguère, Dušan Kereš
Nature (2015-09-23) https://doi.org/f7r445
DOI: 10.1038/nature15383 · PMID: 26399829

91. **HYPERION: an open-source parallelized three-dimensional dust continuum radiative transfer code**
TP Robitaille
Astronomy & Astrophysics (2011-12) https://doi.org/ckp63w
DOI: 10.1051/0004-6361/201117150

92. **Ytree: Merger-Tree Toolkit**
Britton Smith, Meagan Lang
Zenodo (2018-02-16) https://doi.org/gcx4x2
DOI: 10.5281/zenodo.1174374

93. **Simulating X-ray Observations with Python**
John ZuHone, Veronica Biffi, Eric Hallman, Scott Randall, Adam Foster, Christian Schmid
Proceedings of the Python in Science Conference (2014) https://doi.org/gpxcx7
DOI: 10.25080/majora-14bd3278-010

94. **Observing simulated galaxy clusters with phox: a novel X-ray photon simulator**
V Biffi, K Dolag, H Böhringer, G Lemson
Monthly Notices of the Royal Astronomical Society (2012-01) https://doi.org/fz38qm
DOI: 10.1111/j.1365-2966.2011.20278.x

95. **Investigating the velocity structure and X-ray observable properties of simulated galaxy clusters with PHOX**
V Biffi, K Dolag, H Böhringer
Monthly Notices of the Royal Astronomical Society (2012-10-29) https://doi.org/f42dgv
DOI: 10.1093/mnras/sts120

96. **Houdini for Astrophysical Visualization**
JP Naiman, Kalina Borkiewicz, AJ Christensen
Publications of the Astronomical Society of the Pacific (2017-04-18) https://doi.org/f96hcf
DOI: 10.1088/1538-3873/aa51b3

Conclusions

Acknowledgments