November 27, 2017
Draft version November 27, 2017
Typeset using LaTeX **modern** style in AASTeX61

THE ASTROPY PROJECT

Astropy Collaboration

Submitted to ApJ

ABSTRACT

The Astropy project supports and fosters the development of open-source and openly-developed `Python` packages that provide commonly-needed functionality to the astronomical community. A key element of the Astropy project is the core package `astropy`, which serves as the foundation for more specialized projects and packages. In this article, we provide an overview of the organization of the Astropy project and summarize key features in the core package as of the recent major release, version 2.0. We then describe the project infrastructure designed to facilitate and support development for a broader ecosystem of inter-operable packages. We conclude with a future outlook of planned new features and directions for the broader Astropy project.

Keywords: Astrophysics - Instrumentation and Methods for Astrophysics — methods: data analysis — methods: miscellaneous

Corresponding author: Astropy Coordination Committee
coordinators@astropy.org

1. INTRODUCTION

All astronomical research makes use of software in some way. Astronomy as a field has thus long supported the development of software tools for astronomical tasks: from scripts that enable individual scientific research to software packages for small collaborations to data reduction pipelines for survey operations. Some software packages are or were supported by large institutions and are intended for a wide range of users. These packages typically provide some level of documentation and user support or training. Other packages are developed by individual researchers or research groups and are then typically used by smaller groups for more domain-specific purposes. Whether for a package meant for wide distribution or for scripts and programs for a specific research project, the implementation of astronomical software can be eased through the use of a library that provides core functionality that is common to many astronomical tasks. The users of such software then also benefit from a community and ecosystem built around a common foundation. The Astropy project has grown to become this community for `Python` astronomy software, and the `astropy` core package a feature-rich `Python` library.

The development of the `astropy` core package began as a largely community-driven effort to standardize core functionality for astronomical software in `Python`. In this way, its genesis differs from but builds upon many substantial and former astronomical software development efforts that were commissioned or initiated through large institutional support, for example IRAF (developed at NOAO; **?**), MIDAS (developed at ESO; **?**), or Starlink (originally developed by a consortium of UK institutions and now maintained by the East Asian Observatory; **??**). More recently, community-driven efforts have seen significant success in the astronomical sciences (**?**).

`Python`[1] is an increasingly popular, general-purpose programming language that is available under a permissive open source software license free of charge for all major operating systems. The programming language has become especially popular in the quantitative sciences, where researchers must simultaneously produce research, perform data analysis and develop software. A large part of this success owes itself to the vibrant community of developers and a continuously-growing ecosystem of tools, web services and stable well-developed packages that enable easier collaboration on software development, easier writing and sharing of software documentation and continuous testing and validation of software. While the programming language provides support for array representation & arithmetic (`numpy`; **?**), a wide variety of functions for scientific computing (`scipy`; **?**), publication-quality plotting (`matplotlib`; **?**), tens of thousands of other high quality and easy to use packages are available which can help with tasks that are not astronomy specific but might be performed in the course of astronomical research, for example, interfacing with databases or statistical inferences. More recently, the development of package managers such as Anaconda[2]

[1] https://www.python.org/
[2] https://anaconda.org/

have streamlined the installation process for most packages significantly reducing the barrier to entry for using many such libraries.

The Astropy project aims to provide an open-source, open-development core package (`astropy`) and an ecosystem of *affiliated packages* that support astronomical functionality in the `Python` programming language. The `astropy` core package is now a feature-rich library of sufficiently general tools and classes that supports the development of more specialized code. An example of such functionality is reading and writing FITS files: it would be time consuming and impractical for multiple groups to implement the FITS standard (?) and maintain software for such a general-purpose need. Another example of such a common task is in dealing with representations of and transformations between astronomical coordinate systems.

The Astropy project aims to develop and provide high quality code and documentation according to the best practices in software development. The project makes use of these tools to do so without central institutional oversight. The first public release of the `astropy` package is described in ?. Since then, the `astropy` package has been used in hundreds of projects, and the scope of the package has grown considerably. At the same time, the community of astronomers contributing to the project has grown tremendously and an ecosystem of packages supporting or affiliated with the `astropy` core has developed. In this paper, we describe the current status of the Astropy community and the `astropy` core package and discuss goals for future development.

We start by describing the way the Astropy project functions and is organized in Section ??. We then describe the main software efforts developed by the Astropy project itself: a core package called `astropy` (Section ??) and several separate packages that help maintain the infrastructure for, e.g., testing and documentation (Section ??). We end with a short vision for the future of Astropy in particular and astronomical software in general in Section ??.

2. ORGANIZATION AND INFRASTRUCTURE

2.1. *Coordination of Astropy*

From its inception, Astropy has required coordination to ensure the project as a whole and its coding efforts are consistent and reasonably efficient. While many `Python` projects adopt a "Benevolent Dictator For Life" (BDFL) model, Astropy has instead opted for a *coordination committee*. This is in part due to the nature of the project as a large-scale collaboration between many contributors with many interests, and in part due to simply the amount of work that needs to get done. For the latter reason, the project has expanded the committee from three to four members starting in 2016.

For resolving disagreements about the `astropy` core package or other Astropy-managed code, the coordination committee primarily acts to work toward consensus, or when consensus is difficult to achieve, generally acts as a "tie-breaker." The com-

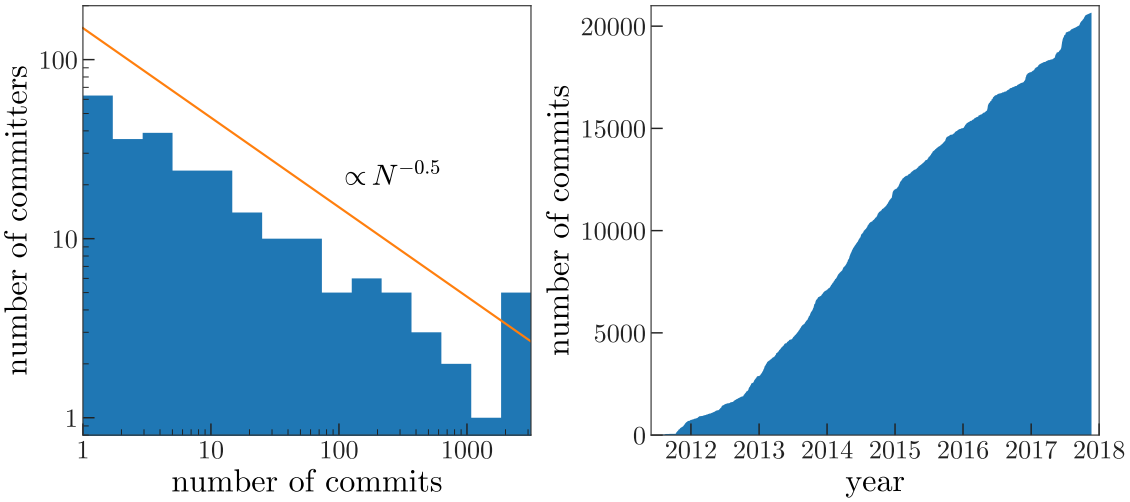

Figure 1. *Left panel*: Distribution of number of commits per committer. *Right panel*: Cumulative number of commits to the `astropy` core package over time.

mittee also oversees affiliated package applications to ensure they are in keeping with Astropy's vision and philosophy, as well as the associated procedures. Additionally, the committee oversees the assignment of roles (primarily driven by already-existing contributions), and increasingly has acted as the "face" of the Project, providing contact with organizations like NumFOCUS (the body that holds any potential funding in trust for Astropy) or the American Astronomical Society (AAS).

2.2. *Astropy development model*

Code is contributed to the `astropy` core package or modified through "pull requests" (via `GitHub`) that often contain several `git` commits. Pull requests may fix bugs, implement new features, or improve or modify the infrastructure that supports the development and maintenance of the package. Individual pull requests are generally limited to a single conceptual addition or modification to make code review tractable. Pull requests that modify or add code to a specific subpackage must be reviewed and approved by one of the subpackage maintainers before it is merged into the core codebase. Bugs and feature requests are reported via the `GitHub` issue tracker and labeled with a set of possible labels that help classify and organize the issues. The development workflow is detailed in the `astropy` documentation.[3]

As of version 2.0, `astropy` contains 212244 lines of code[4] contributed by 232 unique contributors over 19270 `git` commits. Figure **??**, left, shows the distribution of total number of commits per contributor as of November 2017. The relative flatness of this distribution (as demonstrated by its log-log slope of -0.5) shows that the `astropy` core package has been developed by a broad contributor base. A leading group of 6

[3] *How to make a code contribution*, http://docs.astropy.org/en/stable/development/workflow/development_workflow.html

[4] This line count includes comments, as these are often as important for maintainability as the code itself. Without comments there are 142197 lines of code.

developers have each added over 1000 commits to the repository, and ~ 20 more core contributors with at least 100 commits. However, the distribution of contribution level (number of commits) continues from 100 down to a single commit. In this sense, the development of the core package has been a true community effort and is not dominated by a single individual. It is also important to note that the number of commits is only a rough metric of contribution, as a single commit could be a critical fix in the package or a fix for a typographical error. Figure **??**, right, shows the number of contributors as a function of time since the genesis of the `astropy` core package. The package is still healthy: new commits are and have been contributed at a steady rate throughout its existence.

2.3. *APEs - Astropy Proposals for Enhancement*

Central to the success of Astropy is an open environment where anybody can contribute to the project. However, this model leads to "organic" growth where different features are implemented by different people with different programming styles and interfaces. Thus, Astropy has a mechanism to more formally propose significant changes to the core package (e.g., re-writing the coordinates subpackage; **?**), to plan out major new features (e.g., a new file format; **?**), or institute new organization-wide policies (e.g., adopting a code of conduct; **?**). This mechanism is called "Astropy Proposal for Enhancement" (APE) and are modeled after the "Python Enhancement Proposals" (PEP) that guide the development of the `Python` programming language. In an APE, one or more authors describe in detail the proposed changes or additions, including a rationale for the changes, how these changes will be implemented, and, in the case of code, what the interface will be (**?**). The APEs are discussed and refined by the community before much work is invested into a detailed implementation; anyone is welcome to contribute to these discussions during the open consideration period. APEs are proposed via pull requests on a dedicated GitHub repository[5], and anyone can therefore read the proposed APEs and leave comments in-line. In previous APEs a community consensus emerged and APEs are accepted and become the basis for future work at this point. In cases where consensus cannot be reached, the Astropy coordination committee can decide to close the discussion and make an executive decision based on the community input on the APE in question.

2.4. *Concept of affiliated packages*

A major part of the Astropy project is the concept of "Affiliated Packages". An affiliated package is an astronomy-related `Python` package that is not part of the `astropy` core package, but has requested to be included as part of the Astropy project's community. These packages support the goals and vision of Astropy of improving code reuse, interoperability, and embracing good coding practices such as testing and thorough documentation.

[5] https://github.com/astropy/astropy-APEs

Affiliated packages contain functionality that is more specialized, have license incompatibilities, or have external dependencies (e.g., GUI libraries) that make these packages more suitable to be separate from the `astropy` core package. Affiliated packages may also be used to develop substantial new functionality that will eventually be incorporated into the `astropy` core package. New functionality benefits from having a rapid development and release cycle that is not tied to that of the `astropy` core.

Affiliated packages are listed on the main Astropy website and advertised to the community through Astropy mailing lists, so becoming an affiliated package is a good way for new and existing packages to gain exposure while at the same time promoting Astropy's high standard for code and documentation quality. This process of listing and promoting affiliated packages is one way in which the Astropy project tries to increase code reuse in the astronomical community.

Packages can become affiliated to Astropy by applying for this status on a public mailing list. The coordination committee (Section **??**) reviews such requests and issues recommendations for the improvement of a package where possible.

2.5. *Release cycle and Long Term Support*

The `astropy` package has a regular release schedule consisting of new significant releases every 6 months, with bugfix releases as needed (**?**). The major releases contain new features or any significant changes, whereas the bugfix releases only contain fixes to code or documentation but no new features. Some versions are additionally designated as "Long-term support" (LTS) releases, which continue to receive bugfixes for 2 years following the release with no changes to the API. The LTS versions are ideal for pipelines and other applications where API stability is essential. The latest LTS release (v2.0) is also the last one that supports `Python` 2, and will receive bug fixes until the end of 2019 (**?**).

The version numbering of the `astropy` core package reflects this release scheme: the core package version number uses the form x.y.z, where "x" is advanced for LTS releases, "y" is advanced for non-LTS feature releases, and "z" is advanced for bugfix releases.

The released versions of the `astropy` core package are available from several of the `Python` distributions for scientific computing (e.g., Anaconda) and from the `Python` Package Index (PyPI).[6] Effort has been made to make `astropy` available and easily installable across all platforms; the package is constantly tested on different platforms as part of our suite of continuous integration tests.

2.6. *Support of Astropy*

The Astropy project, as of the v2.0 release, does not receive any direct financial support for the development of `astropy`. Development of the software, all supporting

[6] See the installation documentation for more information: http://docs.astropy.org/en/stable/install.html

materials, and community support is provided by individuals who work on the Astropy project in their own personal time, by individuals or groups contributing to Astropy as part of a research project, or contributions from institutions that allocate people to work on Astropy. A list of organizations that have contributed to Astropy in this manner can be found in the Acknowledgements.

Different funding models have been proposed for support of Astropy in the future (e.g., ?), but a long-term plan Different funding models have been proposed for support of Astropy (e.g., ?), but a long-term plan for sustainability has not yet been established. The Astropy project has the ability to accept financial contributions from institutions or individuals through the NumFOCUS[7] organization. NumFOCUS has to date covered the direct costs incurred by the Astropy project.

3. ASTROPY CORE PACKAGE V2.0

The Astropy project aims to provide `Python`-based packages for all tasks that are commonly needed in a large subset of the astronomical community. At the foundation is the `astropy` core package, which provides general functionality (e.g., coordinate transformations, reading and writing astronomical files) or base classes for other packages to utilize for a common interface (e.g., `NDData`). In this section, we highlight new features introduced or substantially improved since version v0.2 (previously described in ?). The `astropy` provides a full log of changes[8] over the course of the entire project and more details about individual subpackages are available in the documentation.[9] Beyond what is mentioned below, most subpackages have seen increased performance since the release of the v0.2 package.

3.1. *Units*

The `astropy.units` subpackage adds support for representing units and numbers with associated units — "quantities" — in code. Historically, quantities in code have often been represented simply as numbers, with units implied or noted via comments in the code because of considerations about speed: having units associated with numbers inherently adds overhead to numerical operations. In `astropy.units`, `Quantity` objects extend `numpy` array object and have been designed with speed in mind.

As of `astropy` version 2.0, units and quantities are prevalent in most other `astropy` subpackages and are thus a key concept for using the package as a whole. Units are intimately entwined in the definition of astronomical coordinates and thus nearly all functionality in the `astropy.coordinates` subpackage (see Section ??) depends on this functionality. For most other subpackages, quantities are at least accepted, and often expected by default.

[7] NumFOCUS is a 501(c)(3) nonprofit that supports and promotes world-class, innovative, open source scientific computing.

[8] https://github.com/astropy/astropy/blob/stable/CHANGES.rst

[9] http://docs.astropy.org/en/stable/

The motivation and key concepts behind this subpackage were described in detail in the previous paper (?), and thus here we primarily highlight new features and improvements.

3.1.1. *Interaction with `numpy` arrays*

The `Quantity` object extends the `numpy.ndarray` object and therefore works well with many of the functions in `numpy` that support array operations. For example, `Quantity` objects with angular units can be directly passed in to the trigonometric functions implemented in `numpy`. The units are internally converted to radians (what the `numpy` trigonometric functions expect) before being passed to `numpy`.

3.1.2. *Logarithmic units and magnitudes*

By default, taking the logarithm of a `Quantity` object with non-dimensionless units intentionally fails. However, some well-known units are actually logarithmic quantities, where the logarithm of the value is taken with respect to some reference value. Examples include astronomical magnitudes, which are logarithmic fluxes, and decibels, which are more generic logarithmic ratios of quantities. Logarithmic, relative units are now supported in `astropy.units`.

3.1.3. *Defining functions that require quantities*

When writing code or functions that expect `Quantity` objects, we often want to enforce that the input units have the correct type. For example, we may want to require only length-type `Quantity` objects. These requirements often lead to implementing repetitive code for validating `Quantity` inputs. `astropy.units` now provides a `Python` decorator, `quantity_input()`, that does this verification automatically.

3.2. *Constants*

The `astropy.constants` subpackage provides a selection of physical and astronomical constants as `Quantity` objects (see Section ??). A brief description of this package was given in ?. In version 2.0, the built-in constants have been organized into modules for specific versions of the constant values. For example, physical constants have `codata2014` (?) and `codata2010` versions. Astronomical constants are organized into `iau2015` and `iau2012` modules to indicate their sources (resolutions from the International Astronomical Union, IAU). The `codata2014` and `iau2015` versions are combined into the default constant value version: `astropyconst20`. For compatibility with `astropy` version 1.3, `astropyconst13` is available and provides access to the adopted versions of the constants from earlier versions of `astropy`. To use previous versions of the constants as *units* (e.g., solar masses), the values have to be imported directly; with version 2.0, `astropy.units` uses the `astropyconst20` versions.

Astronomers using `astropy.constants` should take particular note of the constants provided for Earth, Jupiter, and the Sun. Following IAU 2015 Resolution B3 (?), nominal values are now given for mass parameters and radii. The nominal values will not change even as "current best estimates" are updated.

3.3. *Coordinates*

The `astropy.coordinates` subpackage is designed to support representing and transforming celestial coordinates and, new in version 2.0, velocities. The framework heavily relies on the `astropy.units` subpackage, and most inputs to objects in this subpackage are expected to be `Quantity` objects. Some of the machinery also relies on the Essential Routines of Fundamental Astronomy (ERFA) `C` library for some of the critical underlying transformation machinery (**?**), which is based on the Standards Of Fundamental Astronomy (SOFA) effort (**?**).

A key concept behind the design of this subpackage is that coordinate *representations* and *reference systems / frames* are independent of one another. For example, a set of coordinates in the International Celestial Reference System (ICRS) reference frame could be represented as spherical (right ascension, declination, and distance from solar system barycenter) or Cartesian coordinates (x, y, z with the origin at barycenter). They can therefore change representations independent of being transformed to other reference frames (e.g., the Galactic coordinate frame).

The classes that handle coordinate representations (the `Representation` classes) act like three-dimensional vectors and thus support vector arithmetic. The classes that represent reference systems and frames (the `Frame` classes) internally use `Representation` objects to store the coordinate data—that is, the `Frame` classes accept coordinate data, either as a specified `Representation` object, or using shorthand keyword arguments to specify the components of the coordinates. These preferred representation and short-hand component names differ between various astronomical reference systems. For example, in the ICRS frame, longitude and latitude are right ascension (`ra`) and declination (`dec`), whereas in the Galactic frame, the spherical angles are Galactic longitude (`l`) and latitude (`b`). Each of the `Frame` classes define their own component names and preferred `Representation` class. The frame-specific component names map to corresponding components on the underlying `Representation` object that internally stores the coordinate data. For most frames the preferred representation is spherical, although this is determined primarily by the common use in the astronomical community.

Many of the `Frame` classes also have attributes specific to the corresponding reference system that allow the user to specify the frame. For example, the Fifth Fundamental Catalogue (FK5) reference system requires specifying an equinox to determine the reference frame. If required, these additional frame attributes must be specified along with the coordinate data when a `Frame` object is created. Figure **??** shows the network of possible reference frame transformations as currently implemented in `astropy.coordinates`. Custom, user-implemented `Frame` classes that define transformations to any reference frame in this graph can then be transformed to any of the other connected frames.

The typical user doesn't usually have to interact with the `Frame` or `Representation` classes directly. Instead, `astropy.coordinates` provides a high-level interface to

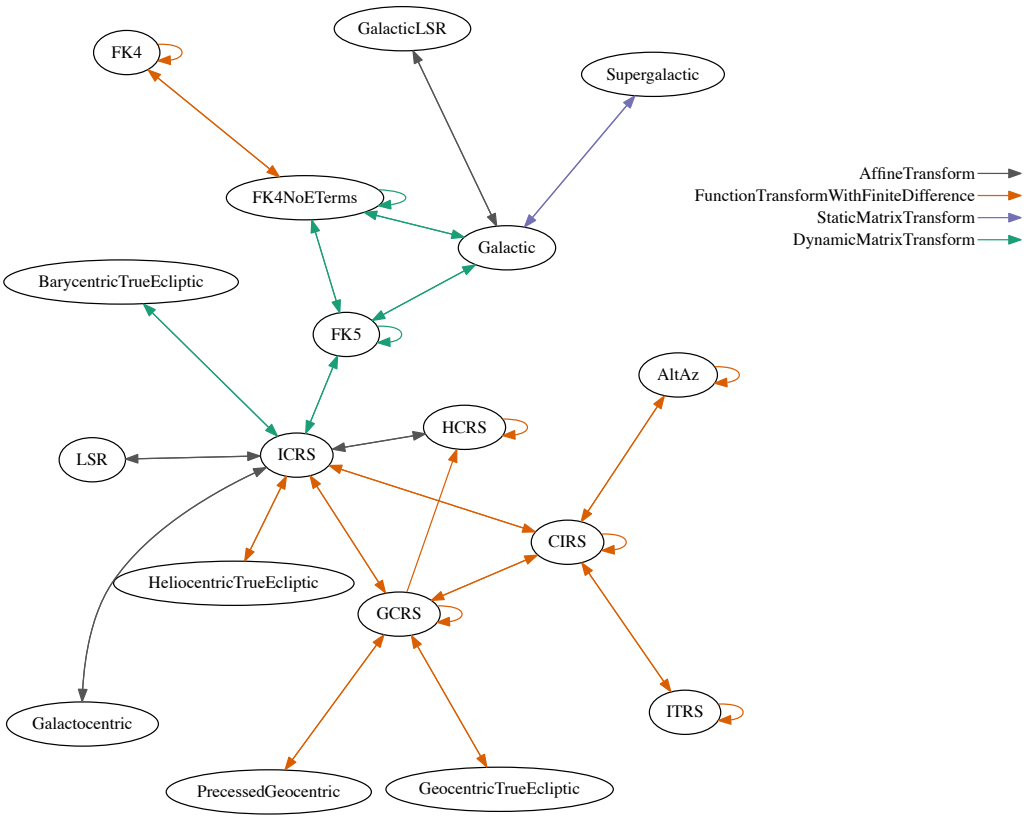

Figure 2. The full graph of possible reference frame transformations implemented in `astropy.coordinates`.

representing astronomical coordinates through the `SkyCoord` class. The `SkyCoord` class was designed to provide a single class that accepts a wide range of possible inputs. It supports coordinate data in any coordinate frame in any representation by internally using the `Frame` and `Representation` classes.

In what follows, we briefly highlight key new features in `astropy.coordinates`.

3.3.1. *Local Earth coordinate frames*

In addition to representing celestial coordinates, `astropy` now supports specifying positions on the Earth in a number of different geocentric systems with the `EarthLocation` class. With this, `astropy` now supports Earth-location-specific coordinate systems such as the altitude-azimuth (`AltAz`) or horizontal system. Transformations between `AltAz` and any Barycentric coordinate frame also requires specifying a time using the `Time` class from `astropy.time`. With this new functionality, many of the common tasks associated with observation planning can now be completed with `astropy` or the Astropy-affiliated package `astroplan` (**?**).

3.3.2. *Proper motion and velocity transformations*

In addition to positional coordinate data, the `Frame` classes now also support velocity data. As the default representation for most frames is spherical, most of the `Frame` classes expect proper motion and radial velocity components to specify the velocity information. The names of the proper motion components all start with `pm` and adopt the same longitude and latitude names as the positional components. Transforming coordinates with velocity data is also supported, but in some cases the transformed velocity components have limited accuracy because the transformations are done numerically. The visualization of the coordinate frame transform graph highlights which velocity transformations can be done exactly and which transformations are done using a finite-difference scheme. The low-level interface for specifying and transforming velocity data (see the next point) is currently experimental. As such, in v2.0, only the `Frame` classes (and not the `SkyCoord` class) support handling velocities.

3.3.3. *Solar System Ephemerides*

Also new is support for computing ephemerides of major solar system bodies and outputting the resulting positions as coordinate objects. These ephemerides can be computed either using analytic approximations from ERFA, or from downloaded JPL ephemerides (the latter requires the `jplephem`[10] optional dependency and an internet connection).

3.3.4. *Accuracy of coordinate transformations*

In order to check the accuracy of the coordinate transformations in `astropy.coordinates`, we have created a set of benchmarks that we use to compare transformations between a set of coordinate frames for a number of packages[11]. Since no package can be guaranteed to implement all transformations to arbitrary precision, and since some transformations are sometimes subject to interpretation of standards (in particular in the case of Galactic coordinates), we do not designate any of the existing packages as the 'ground truth' but instead compare each tool to all other tools to find. The benchmarks are thus useful beyond the Astropy project since they allow all of the tools to be compared to all other tools. The tools included in the benchmark at the moment include the `astropy` core package, Kapteyn (**?**), NOVAS (**?**), PALpy (**?**), PyAST (a wrapper for AST, described in **?**), PyTPM[12], PyEphem (**?**), and pySLALIB (a Python wrapper for SLALIB, described in **?**).

The benchmarks are meant to evolve over time and include an increasing variety of cases. At the moment, the benchmarks are set up as follows - we have generated a standard set of 1000 pairs of random longitudes/latitudes that we use in all benchmarks. Each benchmark is then defined using an input and output coordinate frame, using all combinations of FK4, FK5, Galactic, ICRS and Ecliptic frames. For now we set the epoch of observation to J2000, and the frame to J2000 in the case of the FK5 and ecliptic frames and B1950 for the FK4 frame, but in future we plan to include

[10] https://github.com/brandon-rhodes/python-jplephem
[11] http://www.astropy.org/coordinates-benchmark/summary.html
[12] https://github.com/phn/pytpm

Package 1	Package 2	Median arcsec	Mean arcsec	Maximum arcsec	Std. Dev. arcsec
astropy	kapteyn	0.000	0.000	0.000	0.000
···	palpy	0.218	0.196	0.249	0.056
···	pyast	0.218	0.196	0.249	0.056
···	pyephem	0.459	0.458	0.860	0.231
···	pyslalib	0.218	0.196	0.249	0.056
···	pytpm	0.000	0.000	0.000	0.000
kapteyn	palpy	0.218	0.196	0.249	0.056
···	pyast	0.218	0.196	0.249	0.056
···	pyephem	0.459	0.458	0.860	0.231
···	pyslalib	0.218	0.196	0.249	0.056
···	pytpm	0.000	0.000	0.000	0.000
palpy	pyast	0.000	0.000	0.000	0.000
···	pyephem	0.563	0.570	1.012	0.253
···	pyslalib	0.000	0.000	0.000	0.000
···	pytpm	0.218	0.196	0.249	0.056
pyast	pyephem	0.563	0.570	1.012	0.253
···	pyslalib	0.000	0.000	0.000	0.000
···	pytpm	0.218	0.196	0.249	0.056
pyephem	pyslalib	0.563	0.570	1.012	0.253
···	pytpm	0.459	0.458	0.860	0.231
pyslalib	pytpm	0.218	0.196	0.249	0.056

Table 1. Comparison of the accuracy of the FK4 to Galactic transformation between different packages.

a larger variety of epochs and equinoxes, as well as tests of conversion to/from Altitude/Azimuth. For each benchmark, we convert the 1000 longitudes/latitudes from the input/output frame with all tools and quantify the comparison by looking at the median, mean, maximum, and standard deviation of the absolute separation of the output coordinates from each pair of tools. Table **??** gives an example of the relative accuracy of the conversion from FK4 to Galactic coordinates for all pairs of tools. This shows for example that Astropy, Kapteyn and PyTPM agree perfectly (to the precision shown), while PALpy, pySLALIB, and PyAST also agree perfectly amongst themselves, but show an offset of around 0.2″with the former three packages. Finally, PyEphem disagrees with other packages by 0.4–0.8″. These values are only meant to be illustrative and will change over time as the benchmarks are refined and packages are updated.

3.4. *Time*

The `astropy.time` subpackage focuses on supporting time scales (e.g., UTC, TAI, UT1) and time formats (e.g., Julian date, modified Julian date) that are commonly used in astronomy. This functionality is needed, for example, to calculate barycentric corrections or sidereal times. `astropy.time` is currently built on the ERFA (**?**) C library, which replicates the Standards of Fundamental Astronomy (SOFA; **?**) but is licensed under a three-clause BSD license. The package was described in detail in **?** and has stayed stable for the last several versions of `astropy`. Thus, in what follows, we only highlight significant changes or new features since the previous Astropy paper.

3.4.1. *Barycentric and Heliocentric corrections*

Detailed eclipse or transit timing requires accounting for light travel time differences from the source to the observatory because of the Earth's motion. It is therefore common to instead convert times to the Solar System barycenter or heliocenter where the relative timing of photons is standardized. With the location of a source on the sky (i.e. a `SkyCoord` object), the location of an observatory on Earth (i.e. an `EarthLocation` object), and time values as `Time` objects, the time corrections to shift to the solar system barycenter or heliocenter can now be computed with `astropy.time` using the `light_travel_time` method of a `Time` object.

3.5. *Data containers*

3.5.1. *`nddata`*

The `astropy.nddata` subpackage provides three types of functionality: an abstract interface for representing generic arbitrary-dimensional datasets intended primarily for subclassing by developers of other packages, concrete classes building on this interface, and utilities for manipulating these kind of datasets.

The `NDDataBase` class provides the abstract interface for gridded data with attributes for accessing metadata, the world-coordinate system (WCS), uncertainty arrays matched to the data shape, and other traits. Building on this interface, the `NDData` class provides a minimal working implementation for storing `numpy` arrays. These classes serve as useful base classes for package authors wishing to develop their own classes for specific use cases and as containers for exchanging gridded data.

The classes `NDDataRef`, `NDDataArray`, and `CCDData` extend the base storing functionality with options to do basic arithmetic (addition, subtraction, multiplication, and division) including error propagation in limited cases and slicing of the dataset based on grid coordinates that appropriately handles masking, errors, and units (if present). Additionally, the `CCDData` class also provides reading and writing from and to FITS files and uses data structures from `astropy`, like `WCS`, to represent the contents of a file abstractly.

The `astropy.nddata.utils` module provides utilities that can operate on either plain `numpy` arrays or any of the classes in the `astropy.nddata` subpackage. It features a class for representing two-dimensional image cutouts, allowing one to easily link pixels in the cutout to pixels in the original image or from the image to the

cutout, to convert between world and pixel coordinates in the cutout, and to overlay the cutout on images. Functions to enlarge or reduce an image by doing block replication or reduction are also provided.

3.5.2. *Tables*

The `astropy.table` subpackage provides functionality for representing and manipulating heterogeneous data. The package was described in detail in ?.

Next, we summarize key new features or updates to `astropy.table`.

3.5.3. *Support for grouped table operations*

A table can contain data that naturally forms groups – for example it may contain multiple observations of a few sources at different points in time and in different bands. We may then want to split the table into groups based on the combination of source observed and the band, and then combine the results for each combination of source and band in some way (for example finding the mean or standard deviation of the fluxes or magnitudes over time) or filter the groups based on user-defined criteria. These kinds of grouping and aggregation operations are now fully supported by `Table` objects.

3.5.4. *Support for table concatenation*

`Table` objects can now be combined in several different ways. If two tables have the same columns, we may want to stack them "vertically" to create a new table with the same columns but all rows. If two tables are row-matched but have distinct columns, we may want to stack them "horizontally" to create a new table with the same rows but all columns. For other situations, more general table concatenations or joins are also possible when two tables share some common columns.

3.5.5. *Using astropy objects in tables*

The `Table` object now allows `Quantity` objects, celestial coordinate objects (`SkyCoord`), and date/time objects (`Time`) to be used as columns, and also provides a general way for other user-defined objects to be used as columns. This makes it possible for example to easily represent catalogs of sources or time series as in Astropy, and having both the benefits of the `Table` object (such as accessing specific rows/columns or groups of rows/columns, combining tables, and so on) and the benefits of e.g. the `SkyCoord` or `Time` classes (such as converting the coordinates to a different frame, or accessing the date/time as a modified Julian date or in another time scale).

3.6. *io*

The `astropy.io` subpackage provides support for reading and writing data to a variety of ASCII and binary file formats, such as a wide range of ASCII data table formats, FITS, and VOTable. It also provides a unified interface for reading and writing data with these different formats using the `astropy.table` subpackage. For many common cases this simplifies the process of file input and output and reduces the need to master the separate details of all the I/O packages within `astropy`.

3.6.1. *ASCII*

One of the problems when storing a table in an ASCII format is preserving table meta-data such as comments, keywords and column data types, units, and descriptions. The newly defined *Enhanced Character Separated Values* (ECSV, ?) format makes it possible to write a table to an ASCII-format file and read it back with no loss of information. The ECSV format has been designed to be both human-readable and compatible with most simple CSV readers.

The `astropy.io.ascii` subpackage now includes a significantly faster Cython/C engine for reading and writing ASCII files. This is available for the most common formats. On average the new engine is about 4 to 5 times faster than the corresponding pure-`Python` implementation, and is often comparable to the speed of the `Pandas` (?) ASCII file interface. The fast reader has parallel processing option that allows harnessing multiple cores for input parsing to achieve even greater speed gains. By default, `read()` and `write()` will attempt to use the fast C engine when dealing with compatible formats. Certain features of the full read / write interface are not available in the fast version, in which case the pure-`Python` version will automatically be used.

The `astropy.io.ascii` subpackage now provides the capability to read a table within an HTML file or web URL into an astropy `Table` object. Conversely a `Table` object can now be written out as an HTML table.

3.6.2. *FITS*

The `astropy.io.fits` subpackage started as a direct port of the PyFITS project (?). Therefore it is pretty stable, with mostly bug fixes but also a few new features and performance improvements. The API remains compatible with PyFITS, which is now deprecated in favor of `astropy`.

Command-line scripts are now available for printing a summary of the HDUs in a FITS file(s) (`fitsinfo`) and for printing the header information to the screen in a human-readable format (`fitsheader`).

FITS files are now loaded *lazily* (the default behavior, which can be deactivated if needed), where all HDUs are not loaded until they are requested. This should provide substantial speedups for situations using the convenience functions (e.g., `getheader()` or `getdata()`) to get HDU's that are near the front of a file with many HDUs.

3.7. *Modeling*

3.7.1. *Overview*

The `astropy.modeling` subpackage provides a framework for representing analytical models and performing model evaluation and parameter fitting. Models and fitters are independent of each other: a model can be fit with different fitters and new fitters can be added without changing existing models. The framework is designed to be flexible and easily extensible. The goal is to have a rich set of models but also make

it easy to create new ones if necessary. The modeling framework is used in a variety of data analysis tools and is the basis for the Generalized World Coordinate System (GWCS) package.[13]

3.7.2. *Single Model Definition and Evaluation*

Most models are defined by parameters and maintain an ordered list of parameter names, `Model.param_names`. A model is instantiated by passing in values (scalars or arrays) for its parameters. A parameter is a descriptor that provides a proxy for the value and stores additional information – default value, default unit, and parameter constraints. The value and constraints can be updated by assignment. Supported parameter constraints include `fixed`, and `tied` parameters, and `bounds` on parameter values. Most models have a fixed parameter set but for some (e.g., polynomials), the number of parameters is defined by another argument (e.g., degree-of-freedom in the case of polynomials). Parameters support arithmetic operations and are combined with the inputs during evaluation using the `numpy` broadcasting rules. A model is evaluated by calling it as a function.

Models have a `Model.inverse` property, which returns the analytical inverse, if available, or raises an exception otherwise. This is a settable property; i.e. a model instance can be assigned as inverse to another model. For example, a polynomial model can be assigned as an inverse to another polynomial model.

Another useful settable property of models is `Model.bounding_box`. This attribute sets the domain over which the model is defined. This greatly improves the efficiency of evaluation when the input range is much larger than the characteristic width of the model itself.

3.7.3. *Model Sets*

`astropy.modeling` provides an efficient way to instantiate the same type of model with many different sets of parameter values by passing the `n_models` argument on instantiation, which sets the number of models to instantiate. This creates a model set that can be efficiently evaluated for example, in PSF (point spread function) photometry, all objects in an image will have a PSF of the same functional form, but with different positions and amplitudes.

3.7.4. *Compound Models*

Models can be combined using arithmetic expressions. The result is also a model, which can further be combined with other models. Modeling supports arithmetic (+, -, *, /, and **), join (&), and composition (|) operators. The rules for combining models involve matching their inputs and outputs. For example, the composition operator, |, requires the number of outputs of the left model to be equal to the number of inputs of the right one. For the join operator, the total number of inputs must equal

[13] https://github.com/spacetelescope/gwcs

the sum of number of inputs of both the left and the right models. For all arithmetic operators, the left and the right models must have the same number of inputs and outputs. An example of a compound model could be a spectrum with interstellar absorption. The stellar spectrum and the interstellar extinction are represented by separate models, but the observed spectrum is fitted with a compound model that combines both.

3.7.5. *Fitting Models to Data*

`astropy.modeling` provides several fitters which are wrappers around some of the `numpy` and `scipy.optimize` functions and provide support for specifying parameter constraints. The fitters take a model and data as input and return a copy of the model with the optimized parameter values set. The goal is to make it easy to extend the fitting framework to create new fitters. The optimizers available in `astropy` version 2.0 are Levenberg–Marquardt, Simplex, SLSQP, and LinearLSQFitter (which is based on `numpy.linalg` that provides exact solution for linear models).

Modeling also supports a plugin system for fitters, which allows using the `astropy` models with external fitters. An example of this is `SABA`[14], which is a bridge between Sherpa[15] and `astropy.modeling`, to bring the Sherpa fitters into `astropy`.

3.7.6. *Creating New Models*

New model classes can be created in two ways:

1. The simplest way is to use the `custom_model` decorator in modeling with a user-defined function that takes the inputs and model parameters as arguments.
2. It is also possible to create an arbitrarily complex custom model based on the `Model` class.

3.7.7. *Unit Support*

The `astropy.modeling` subpackage now supports the representation, evaluation, and fitting of models using `Quantity` objects, which attach units to scalar values or arrays of values. In practice, this means that one can, for example, fit a model to data with units and get parameters that also have units out, or initialize a model with parameters with units and evaluate it using input values with different but equivalent units. Using this, we have implemented a blackbody model (`BlackBody1D`) that can be used to fit observed fluxes in a variety of units and as a function of different units of spectral coordinates (e.g., wavelength or frequency).

3.8. *Convolution*

The `astropy.convolution` subpackage implements 'normalized convolution' (e.g., **?**), which is an image reconstruction technique in which missing data are ignored

[14] https://github.com/astropy/saba
[15] http://cxc.cfa.harvard.edu/contrib/sherpa/

during the convolution and replaced with values interpolated using the kernel. An example of this is given in Figure **??**. In versions ≤ 1.3, the direct convolution and Fast Fourier Transform (FFT) convolution approaches were not consistent, with FFT convolution implementing normalized convolution and direct convolution implementing a different approach. As of version 2.0, the two methods are consistent and include a suite of consistency checks.

3.9. *Visualization*

The `astropy.visualization` subpackage provides functionality that can be helpful when visualizing data. This includes a framework for plotting astronomical images with coordinates with `matplotlib` (previously the standalone `wcsaxes` package), functionality related to image normalization (including both scaling and stretching), smart histogram plotting, RGB color image creation from separate images, and custom plotting styles for `matplotlib`.

3.9.1. *Image Stretching and Normalization*

`astropy.visualization` provides a framework for transforming values in images (and more generally any arrays), typically for the purpose of visualization. Two main types of transformations are normalization and stretching of image values.

Normalization transforms the images values to the range $[0, 1]$ using lower and upper limits $(v_{\min}, v_{\max})$

$$y = \frac{x - v_{\min}}{v_{\max} - v_{\min}} \tag{1}$$

where x represents the values in the original image.

Stretching transforms the image values in the range $[0, 1]$ again to the range $[0, 1]$ using a linear or non-linear function

$$z = f(y) \quad . \tag{2}$$

Several classes are provided for automatically determining intervals (e.g., using image percentiles) and for normalizing values in this interval to the $[0, 1]$ range.

`matplotlib` allows a custom normalization and stretch to be used when displaying data by passing a normalization object. The `astropy.visualization` package also provides a normalization class that wraps the interval and stretching objects into a normalization object that `matplotlib` understands.

3.9.2. *Plotting image data with world coordinates*

Astronomers dealing with observational imaging commonly need to make figures with images that include the correct coordinates and optionally display a coordinate grid. The challenge, however, is that the conceptual coordinate axes (such as longitude/latitude) need not be lined up with the pixel axes of the image. The `astropy.visualization.wcsaxes` subpackage implements a generalized way of making figures from an image array and a world coordinate system (WCS) object that provides the transformation between pixel and 'world' coordinates.

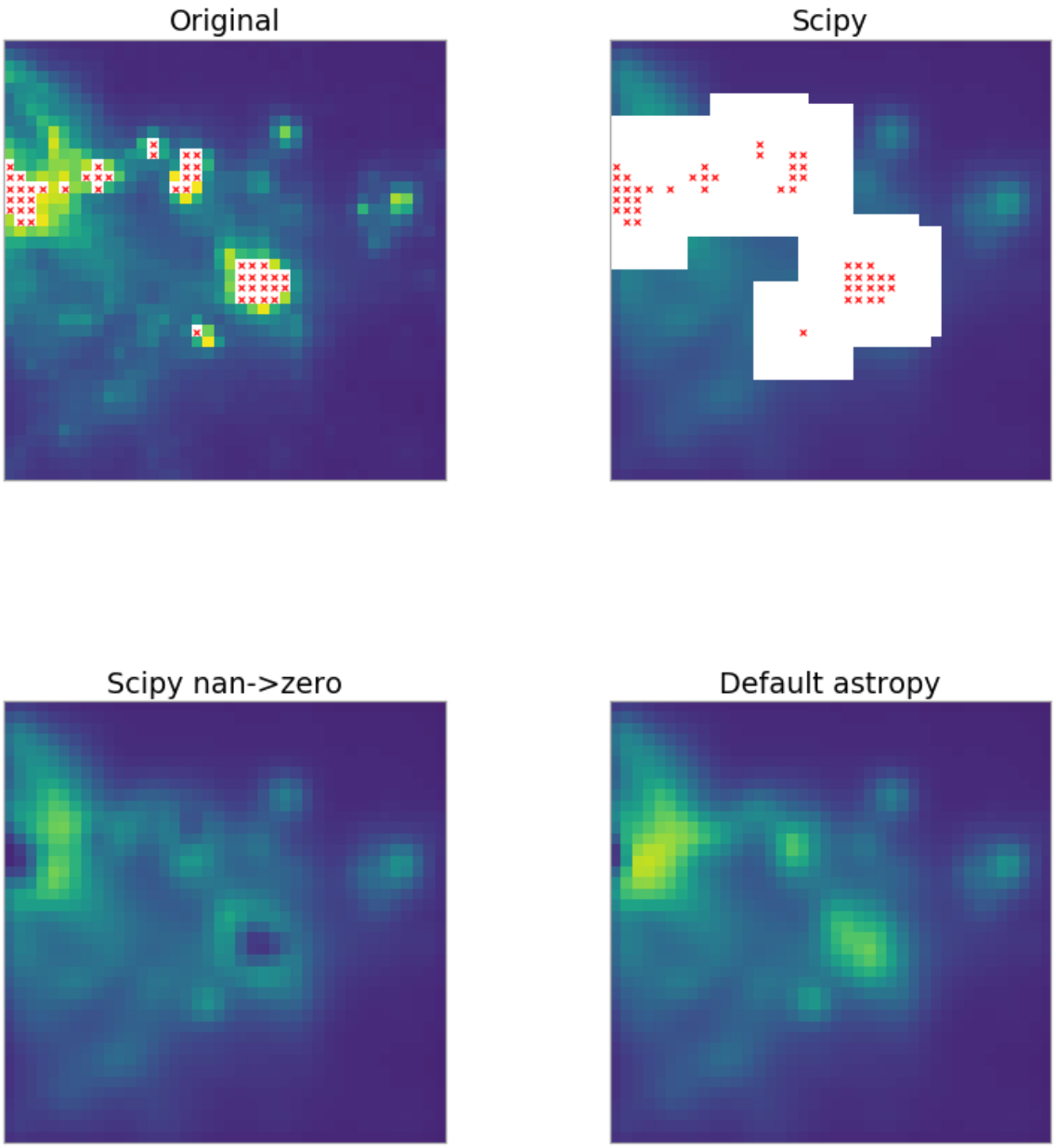

Figure 3. An example showing different modes of convolution available in the `Python` ecosystem. The red x's mark pixels that are set to NaN in the original data (a). If the data are convolved with a Gaussian kernel on a 9×9 grid using scipy's direct convolution (b), any pixel within range of the original NaN pixels is also set to NaN. Panel (c) shows what happens if the NaNs are set to zero first: the originally NaN regions are depressed relative to their surroundings. Finally, panel (d) shows `astropy`'s convolution behavior, where the missing pixels are replaced with values interpolated from their surroundings using the convolution kernel.

World coordinates can be, for example, right ascension and declination, but can also include, for example, velocity, wavelength, frequency, or time. The main features from this subpackage include the ability to control which axes to show which coordinate on (for example showing longitude ticks on the top and bottom axes and latitude on the left and right axes), controlling the spacing of the ticks either by specifying the

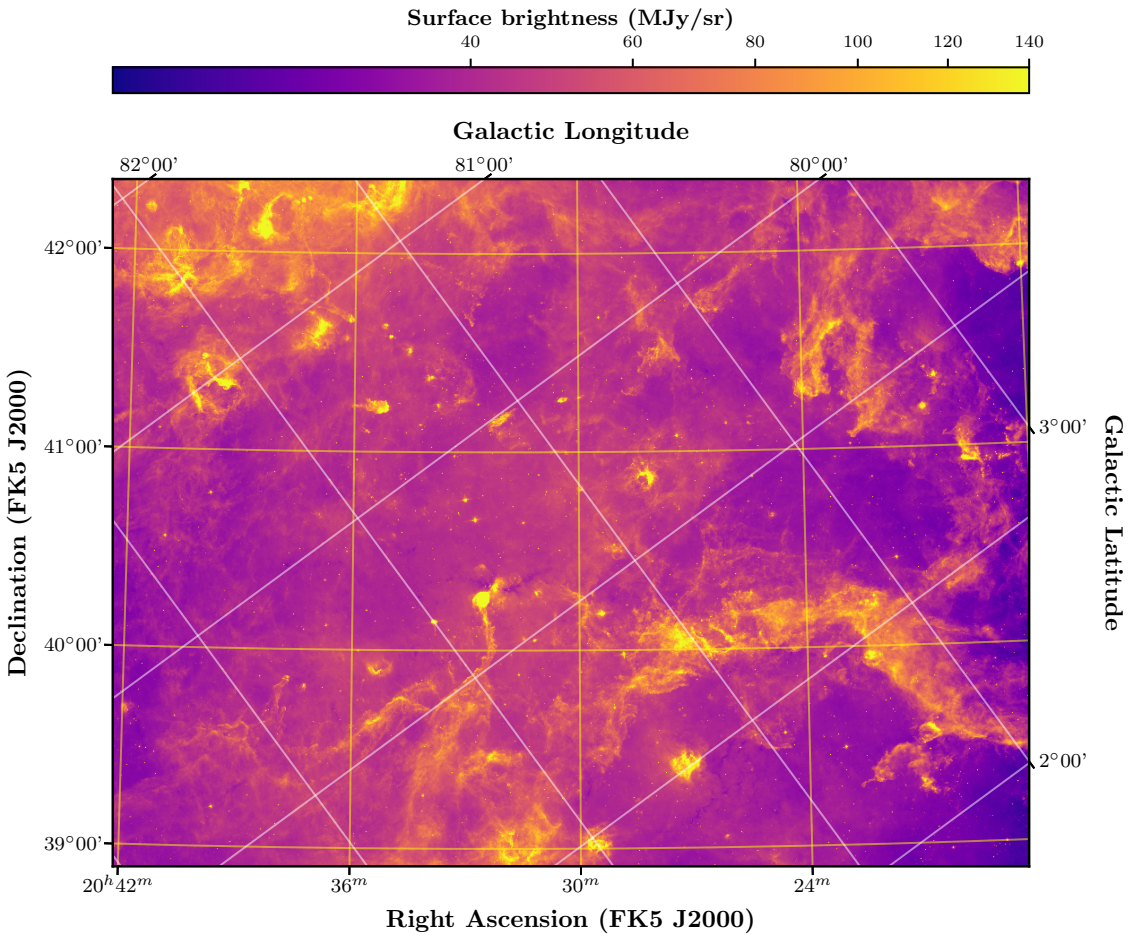

Figure 4. An example of figure made using the `astropy.visualization.wcsaxes` subpackage, using *Spitzer*/IRAC 8.0 .0μm data from the Cygnus-X *Spitzer* Legacy survey (**?**).

positions to use or providing a tick spacing or an average number of ticks that should be present on each axis, setting the format for the tick labels to ones commonly used by astronomers, controlling the visibility of the grid/graticule, and overlaying ticks, labels, and/or grid lines from different coordinate systems. In addition, it is possible to pass data with more than two dimensions and slice on-the-fly. Finally, it is possible to define non-rectangular frames, such as, for example, Aitoff projections.

This subpackage differs from `APLpy` (**?**) in that the latter focuses on providing a very high-level interface to plotting that requires very few lines of code to get a good result, whereas `wcsaxes` defines an interface that is much closer to that of `matplotlib` (**?**). This enables significantly more advanced visualizations.

An example of a visualization made with `wcsaxes` is shown in Figure **??** – this example illustrates the ability to overlay multiple coordinate systems and customize which ticks/labels are shown on which axes around the image. This also uses the image stretching functionality from Section **??** to show the image in a square root stretch (automatically updating the tick positions in the colorbar).

3.9.3. *Choosing Histogram Bins*

`astropy.visualization` also provides a histogram function, which is a generalization of `matplotlib`'s histogram function, to allow for more flexible specification of histogram bins. The function provides several methods of automatically tuning the histogram bin size. It has a syntax identical to `matplotlib`'s histogram function, with the exception of the `bins` parameter, which allows specification of one of four different methods for automatic bin selection: 'blocks', 'knuth', 'scott', or 'freedman'.

3.9.4. *Creating color RGB images*

? describe an "optimal" algorithm for producing red-green-blue (RGB) composite images from three separate high-dynamic range arrays. The `astropy.visualization` subpackage provides a convenience function to create such a color image. It also includes an associated set of classes to provide alternate scalings. This functionality was contributed by developers from the Large Synoptic Survey Telescope (LSST) and serves as an example of contribution to Astropy from a more traditional engineering organization (**?**).

The Sloan Digital Sky Survey (SDSS) SkyServer color images were made using a variation on this technique. As an example, in Figure **??** we show an RGB color image of the Hickson 88 group, centered near NGC 6977. This image was generated from SDSS images using the `astropy.visualization` tools.

3.10. *Cosmology*

The `cosmology` package contains classes for representing different cosmologies and functions for calculating commonly used quantities such as look-back time and distance. The package was described in detail in **?**. The default cosmology in Astropy version 2.0 is given by the values in **?**.

3.11. *Statistics*

The `astropy.stats` package provides statistical tools that are useful for or specific to astronomy and are not found in or extend the available functionality of other `Python` statistics packages such as `scipy` (**?**) or `statsmodels` (**?**). `astropy.stats` contains a range of functionality used by many different disciplines in astronomy. It is not a complete set of statistic tools, but rather a still growing collection of useful features.

In this section, we describe these tools, including robust statistical estimators, circular statistics, periodograms, spatial statistics, and histogram binning.

3.11.1. *Robust Statistical Estimators*

Robust statistics provide reliable estimates of basic statistics for complex distributions that largely mitigate the effects of outliers. `astropy.stats` includes several robust statistical functions that are commonly used in astronomy, such as sigma clipping methods for rejecting outliers, median absolute deviation functions, and biweight

Figure 5. An RGB color image of the region near the Hickson 88 group constructed from SDSS images and the `astropy.visualization` tools.

estimators, which have been used to calculate the velocity dispersion of galaxy clusters (**?**).

3.11.2. *Circular Statistics*

A set of circular statistical estimators based on **?** are implemented in `astropy.stats`. These functions provide measurements of the circular mean, variance, and moment. For all of these functions, they work with both `numpy.ndarrays` (assumed to be in radians) and `Quantity` objects. In addition, the subpackage includes tests for Rayleigh Test, vtest, and a function to compute the maximum likelihood estimator for the parameters of the von Mises distribution.

3.11.3. *Lomb-Scargle Periodograms*

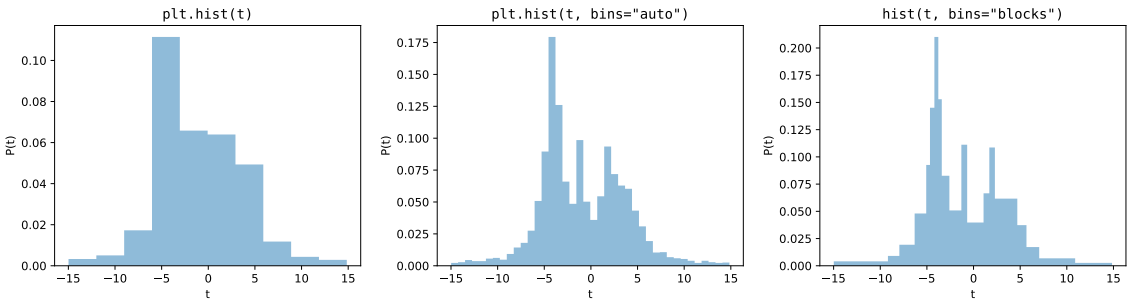

Figure 6. Three approaches to a 1D histogram: *left:* a standard histogram using `matplotlib`'s default of 10 bins. *center:* a histogram with the number of equal-width bins determined automatically using `numpy`'s `bins='auto'`. *right:* a histogram created with `astropy`, with irregularly-spaced bins computed via the Bayesian Blocks algorithm. Compared to regularly-spaced bins, the irregular bin widths give a more accurate visual representation of features in the dataset at various scales.

Periodic analysis of unevenly-spaced time series is common across many subfields of astronomy. The `astropy.stats` package now includes several efficient implementations of the Lomb-Scargle periodogram (**??**) and several generalizations, including floating mean models (**?**), truncated Fourier models (**?**), and appropriate handling of heteroscedastic uncertainties. Importantly, the implementations make use of several fast and scalable computational approaches (e.g., **??**), and so can be applied to much larger datasets than Lomb-Scargle algorithms available in, e.g., `scipy.stats` (**?**). Much of the Lomb-Scargle code in Astropy has been adapted from previously-published open-source code (**??**).

3.11.4. *Bayesian Blocks and Histogram Binning*

`astropy` also includes an implementation of *Bayesian Blocks* (**?**), an algorithm for analysis of break-points in non-periodic astronomical time-series. One interesting application of Bayesian Blocks is its use in determining optimal histogram binnings, and in particular binnings with unequal bin sizes. This code was adapted, with several improvements, from the `astroML` package (**?**). An example of a histogram fit using the Bayesian blocks algorithm is shown in the right panel of Figure **??**.

4. INFRASTRUCTURE FOR ASTROPY AFFILIATED PACKAGES

In addition to astronomy-specific packages and libraries, the Astropy Project also maintains and distributes several general-purpose infrastructure packages that help with the maintenance and upkeep of the `astropy` core package and other affiliated packages. The following sections describes the most widely-used infrastructure packages developed by the Astropy Project.

4.1. *Package template*

Astropy provides a package template — as a separate `GitHub` repository, `astropy/package-template`[16] — that aims to simplify setting up packaging, testing, and documentation builds for developers of affiliated packages or `astropy`-dependent packages. Any `Python` package can make use of this ready-to-go package layout, setup, installation, and `Sphinx` documentation build infrastructure that was originally developed for the `astropy` core package and affiliated packages maintained by the Astropy project. The package template also provides a testing framework, template configurations for continuous integration services, and `Cython` build support.

4.2. *Continuous integration helpers*

Astropy also provides a set of scripts for setting up and configuring continuous integration (CI) services as a `GitHub` repository, `astropy/ci-helpers`.[17] These tools aim to empower package maintainers to control their testing set up and installation process for various continuous integration services through a set of environment variables. While the current development is mostly driven by the needs of the Astropy ecosystem, the actual usage of this package is extremely widespread. The current tools support configuration for Travis CI and Appveyor CI.

4.3. *Sphinx extensions*

The documentation for many `Python` packages, the core `astropy` package, and for all packages in the Astropy ecosystem is written using the `Sphinx` documentation build system. `Sphinx` makes it possible to write documentation using plain text files that follow a markup language called "reStructuredText," which are then transformed into HTML or LaTeX documentation during the documentation build process. For the Astropy project, we have developed a few `Sphinx` extensions that facilitate automatically generating API documentation for large projects, like the `astropy` core package. The main extension we have developed is `sphinx-automodapi`[18], which makes it easy with a single reStructuredText command to generate a set of documentation pages listing all of the available classes, functions, and attributes in a given `Python` module.

5. THE FUTURE OF THE ASTROPY PROJECT

Following the release of version 2.0, development on the next major version of the `astropy` core package (version 3.0) has already begun. On top of planned changes and additions to the core package, we also plan to overhaul the Astropy educational and learning materials, and further refactor and generalize the infrastructure utilities developed for the core package for the benefit of the community.

5.1. *Future versions of the* `astropy` *core and affiliated packages*

One of the most significant changes coming in this next major release will be removing support for `Python` 2 (`?`): future versions of `astropy` will only support `Python`

[16] https://github.com/astropy/package-template/
[17] https://github.com/astropy/ci-helpers
[18] http://sphinx-automodapi.readthedocs.io

3.5 and higher. Removing `Python` 2 support will allow the use of new, `Python` 3-only features, will simplify the code base, and reduce the testing overhead for the package. `astropy` version 3.0 is currently scheduled for January 2018.

In the next major release after version 3.0, scheduled for the summer of 2018, the focus will be on optimization of the code and improved documentation. To prepare for this release, we are preparing software for testing, evaluating, and monitoring the performance of the code. Less functionality may be introduced in this release while the focus will be primarily on improved performance.

Beyond the core code, the Astropy project is also further developing the Astropy-managed affiliated packages. While these may not be integrated into the `astropy` core package, these projects provide code that is useful to the astronomical community and meet the testing and documentation standards of Astropy. Some of these new efforts includes an initiative to develop tools for spectroscopy (`?`, `specutils`, `specreduc`, `specviz`), integration of LSST software, and packages to support HEALPIX projection.

5.2. *Learn Astropy*

The `astropy` core package documentation contains narrative descriptions of the package functionality along with detailed usage notes for functions, classes, and modules. While useful as a reference and for more advanced `Python` users, it is not the right entry-point for all users or learning environments. In the near future, we will launch a new resource for learning to use both the `astropy` core package and the many packages in the broader Astropy ecosystem, under the name '*Learn Astropy*.'

The new *Learn Astropy* site will present several different ways to engage with the Astropy ecosystem:

Documentation: The `astropy` and affiliated package documentation contain the complete description of a package with all requisite details, including usage, dependencies, and examples. The pages will largely remain as-is, but will be focused towards more intermediate users and as a reference resource.

Examples: These are stand-alone code snippets that live in the `astropy` documentation that demonstrate a specific functionality within a subpackage. The `astropy` core package documentation will then gain a new "index of examples" that links to all of the code or demonstrative examples within any documentation page.

Tutorials: The Astropy tutorials are step-by-step demonstrations of common tasks that incorporate several packages or subpackages. Tutorials are more extended and comprehensive than examples, may contain exercises for the users, and are generally geared towards workshops or teaching. Several tutorials already exist[19] and are being actively expanded.

[19] http://tutorials.astropy.org/

Guides: These are long-form narrative, comprehensive, conceptually-focused documents (roughly one book chapter in length) providing stand-alone introductions to core packages in addition to the underlying astronomical concepts. These are less specific and more conceptual than tutorials. For example, "using `astropy` and `ccdproc` to reduce imaging data."

We encourage any users who wish to see specific material to either contribute or comment on these efforts via the Astropy mailing list or Astropy-tutorials `GitHub` repository.[20]

6. CONCLUSION

The development of the `astropy` package and cultivation of the Astropy ecosystem is still maintaining significant growth while improving stability, breadth, and reliability. As the `astropy` core package becomes more mature, several subpackages have reached stability with a rich set of features that help astronomers worldwide to perform many daily tasks such as planning observations, analyzing data or simulation results, and writing publications. The strong emphasis that the Astropy project puts on reliability and code correctness helps users to trust the calculations performed with `astropy` and to publish reproducible results. At the same time, the Astropy ecosystem and core package are growing: new functionality is still being contributed, and new affiliated packages are being developed to support more specialized needs.

The Astropy project is also spreading awareness of best practices in community-driven software development. This is important as most practicing astronomers were not explicitly taught computer science and software development, despite the fact that a major fraction of nearly every astronomer's workload today is related to software use and development. The `astropy` package leads by example, showing all interested astronomers how modern tools like `git` version control or continuous integration can increase the quality, accessibility, and discoverability of astronomical software without overly complicating the development cycle. Within Astropy, all submitted code is reviewed by at least one, but typically more, members of the Astropy community, who provide feedback to contributors to help improve their skills. As a community, we follow an explicit code of conduct (**?**) and treat all contributors and users with respect, provide a harassment-free environment, and encourage and welcome new contributions from all. Thus, while the Astropy project provides and develops software and tools essential to modern astronomical research, it also helps to prepare the current and next generation of researchers with the knowledge to adequately use, develop, and contribute to those tools within a conscientious and welcoming community.

Who to thank? numfocus, GSOC, Python Software Foundation

We thank the many web services that make developing and maintaining this package feasible, such as GitHub, Travis CI, Appveyor, CircleCI, and Read the Docs.

[20] https://github.com/astropy/astropy-tutorials

Software: `astropy` (?), `numpy` (?), `scipy` (?), `matplotlib` (?), `Cython` (?),

APPENDIX

A. LIST OF AFFILIATED PACKAGES

Table 2. Registry of affiliated packages.

Package Name	Stable	PyPI Name	Maintainer	Citation
Astro-SCRAPPY	Yes	astroscrappy	Curtis McCully	?
astroplan	No	astroplan	Brett Morris	?
astroquery	Yes	astroquery	Adam Ginsburg and Brigitta Sipocz	?
ccdproc	Yes	ccdproc	Steven Crawford, Matt Craig, and Michael Seifert	?
cluster-lensing	No	cluster-lensing	Jes Ford	?
gala	Yes	astro-gala	Adrian Price-Whelan	?
galpy	Yes	galpy	Jo Bovy	?
gammapy	No	gammapy	Christoph Deil	?
ginga	Yes	ginga	Eric Jeschke and Pey-Lian Lim	?
Glue	Yes	glueviz	Chris Beaumont and Thomas Robitaille	?
gwcs	No	gwcs	Nadia Dencheva	?
Halotools	Yes	halotools	Andrew Hearin	?
imexam	No	imexam	Megan Sosey	?
marxs	Yes	marxs	Hans Moritz Günther (hamogu)	?
montage-wrapper	Yes	montage-wrapper	Thomas Robitaille	
naima	Yes	naima	Victor Zabalza	?
photutils	No	photutils	Larry Bradley and Brigitta Sipocz	?
PyDL	No	pydl	Benjamin Alan Weaver	
python-cpl	No	python-cpl	Ole Streicher	?

Table 2 continued on next page

Table 2 *(continued)*

Package Name	Stable	PyPI Name	Maintainer	Citation
reproject	Yes	reproject	Thomas Robitaille	
sncosmo	Yes	sncosmo	Kyle Barbary	?
spectral-cube	Yes	spectral-cube	Adam Ginsburg	?
specutils	No	specutils	Nicholas Earl, Adam Ginsburg, Steve Crawford	

Table 3. Registry of provisionally accepted affiliated packages.

Package Name	Stable	PyPI Name	Maintainer
APLpy	Yes	APLpy	Thomas Robitaille and Eli Bressert
astroML	Yes	astroML	Jake Vanderplas
HENDRICS	Yes	hendrics	Matteo Bachetti
omnifit	Yes	omnifit	Aleksi Suutarinen
pyregion	Yes	pyregion	Jae-Joon Lee
PyVO	No	pyvo	Stefan Becker
spherical_geometry	No	spherical-geometry	Bernie Simon and Michael Droettboom