The Journal of Open Source Software

unyt: Handle, manipulate, and convert data with units in Python

Nathan J. Goldbaum[1], **John A. ZuHone**[2], **Matthew J. Turk**[1], **Kacper Kowalik**[1], **and Anna L. Rosen**[2]

1 National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign. 1205 W Clark St, Urbana, IL USA 61801 **2** Harvard-Smithsonian Center for Astrophysics. 60 Garden St, Cambridge, MA USA 02138

DOI: 10.21105/joss.00809

Software

- Review
- Repository
- Archive

Submitted: 06 June 2018
Published: 14 August 2018

Summary

Software that processes real-world data or that models a physical system must have some way of managing units. This might be as simple as the convention that all floating point numbers are understood to be in the same physical unit system (for example, the SI MKS units system). While simple approaches like this do work in practice, they also are fraught with possible error, both by programmers modifying the code who unintentionally misinterpret the units, and by users of the software who must take care to supply data in the correct units or who need to infer the units of data returned by the software. Famously, NASA lost contact with the Mars Climate Orbiter spacecraft after it crash-landed on the surface of Mars due to the use of English Imperial units rather than metric units in the spacecraft control software (Board, 1999).

The `unyt` library is designed both to aid quick calculations at an interactive Python prompt and to be tightly integrated into a larger Python application or library. The top-level `unyt` namespace ships with a large number of predefined units and physical constants to aid setting up quick calculations without needing to look up unit data or the value of a physical constant. Using the `unyt` library as an interactive calculation aid only requires knowledge of basic Python syntax and awareness of a few of the methods of the `unyt_array` class — for example, the `unyt_array.to()` method to convert data to a different unit. As the complexity of the usage increases, `unyt` provides a number of optional features to aid these cases, including custom unit registries containing both predefined physical units as well as user-defined units, built-in output to disk via the pickle protocol and to HDF5 files using the `h5py` library (Collette, 2013), and round-trip converters for unit objects defined by other popular Python unit libraries.

Physical units in the `unyt` class are defined in terms of the dimensions of the unit, a string representation, and a floating point scaling to the MKS unit system. Rather than implementing algebra for unit expressions, we rely on the `SymPy` symbolic algebra library (Meurer et al., 2017) to handle symbolic algebraic manipulation. The `unyt.Unit` object can represent arbitrary units formed out of base dimensions in the SI unit system: time, length, mass, temperature, luminance, and electric current. We currently treat units such as mol with the seventh SI base dimension, amount of substance, as dimensionless, although we are open to changing this based on feedback from users. The `unyt` library supports forming quantities defined in other unit systems — in particular CGS Gaussian units common in astrophysics as well as geometrized "natural" units common in relativistic calculations. In addition, `unyt` ships with a number of other useful predefined unit systems including imperial units; Planck units; a unit system for calculations in the solar system; and a "galactic" unit system based on the solar mass, kiloparsecs, and Myr, a convention common in galactic astronomy.

In addition to the `unyt.Unit` class, `unyt` also provides a two subclasses of the NumPy (Oliphant, 2006) `ndarray` (Walt, Colbert, & Varoquaux, 2011), `unyt.unyt_array` and `unyt.unyt_quantity` to represent arrays and scalars with units attached, respectively. The `unyt` library also provides a `unyt.UnitRegistry` class to allow custom systems of units, for example to track the internal unit system used in a simulation. These subclasses are tightly integrated with the NumPy ufunc system, which ensures that algebraic calculations that include data with units automatically check to make sure the units are consistent, and allow automatic converting of the final answer of a calculation into a convenient unit.

We direct readers interested in usage examples and a guide for integrating `unyt` into an existing Python application or workflow to the unyt documentation hosted at http://unyt.readthedocs.io/en/latest/.

Comparison with `Pint` and `astropy.units`

The scientific Python ecosystem has a long history of efforts to develop a library to handle unit conversions and enforce unit consistency. For a relatively recent review of these efforts, see (Bekolay, 2013). While we won't exhaustively cover extant Python libraries for handling units in this paper, we will focus on `Pint` (Grecco, 2018) and `astropy.units` (The Astropy Collaboration et al., 2018), which both provide a robust implementation of an array container with units attached and are commonly used in research software projects. At time of writing a GitHub search for `import astropy.units` returns approximately 10,500 results and a search for `import pint` returns approximately 1,500 results.

While `unyt` provides functionality that overlaps with `astropy.units` and `Pint`, there are important differences which we elaborate on below. In addition, it is worth noting that all three codebases had origins at roughly the same time period. `Pint` initially began development in 2012 according to the git repository logs. Likewise `astropy.units` began development in 2012 and was released as part of `astropy 0.2` in 2013, although the initial implementation was adapted from the `pynbody` library (Pontzen et al., 2013), which started its units implementation in 2010 according to the git repository logs. In the case of `unyt`, it originated via the `dimensionful` library (Stark, 2012) in 2012. Later, `dimensionful` was elaborated on and improved to become `yt.units`, the unit system for the `yt` library (Turk et al., 2011) at a `yt` developer workshop in 2013 and was subsequently released as part of `yt 3.0` in 2014. One of the design goals for the `yt` unit system was the ability to dynamically define "code" units (e.g. units internal to data loaded by yt) as well as units that depend on details of the dataset — in particular cosmological comoving units and the "little h" factor, used to parameterize the Hubble constant in cosmology calculations (Croton, 2013). For cosmology simulations in particular, comparing data with different unit systems can be tricky because one might want to use data from multiple outputs in a time series, with each output having a different mapping from internal units to physical units. This despite the fact that each output in the time series represents the same physical system and common workflows involve combining data from multiple outputs. This requirement to manage complex custom units and interoperate between custom unit systems drove the `yt` community to independently develop a custom unit system solution. We have decided to repackage and improve `yt.units` in the form of `unyt` to both make it easier to work on and improve the unit system and encourage use of the unit system for scientific Python users who do not want to install a heavy-weight dependency like `yt`.

Below we present a table comparing `unyt` with `astropy.units` and `Pint`. Estimates for lines of code in the library were generated using the `cloc` tool (Danial, 2018); blank and comment lines are excluded from the estimate. Test coverage was estimated using the

coveralls output for Pint and astropy.units and using the codecov.io output for unyt.

Library	unyt	astropy.units	Pint
Lines of code	5128	10163	8908
Lines of code excluding tests	3195	5504	4499
Test Coverage	99.91%	93.63%	77.44%

We offer lines of code as a very rough estimate for the "hackability" of the codebase. In general, smaller codebases with higher test coverage have fewer defects (Gopinath, Jensen, & Groce, 2014; Koru, Zhang, & Liu, 2007; Lipow, 1982). This comparison is somewhat unfair in favor of unyt in that astropy.units only depends on NumPy and Pint has no dependencies, while unyt depends on both SymPy and NumPy. Much of the reduction in the size of the unyt library can be attributed to offloading the handling of algebra to SymPy rather than needing to implement the algebra of unit symbols directly in unyt. For potential users who are wary of adding SymPy as a dependency, that might argue in favor of using Pint in favor of unyt.

astropy.units

The astropy.units subpackage provides a PrefixUnit class, a Quantity class that represents both scalar and array data with attached units, and a large number of predefined unit symbols. The preferred way to create Quantity instances is via multiplication with a PrefixUnit instance. Similar to unyt, the Quantity class is implemented via a subclass of the NumPy ndarray class. Indeed, in many ways the everyday usage patterns of astropy.units and unyt are similar, although unyt is not quite a drop-in replacement for astropy.units as there are some API differences. The main functional difference between astropy.units and unyt is that astropy.units is a subpackage of the larger astropy package. This means that depending on astropy.units requires installing a large collection of astronomically focused software included in the astropy package, including a substantial amount of compiled C code. This presents a barrier to usage for potential users of astropy.units who are not astronomers or do not need the observational astronomy capabilities provided by astropy.

Pint

The Pint package provides a different API for accessing units compared with unyt and astropy.units. Rather than making units immediately importable from the Pint namespace, Pint instead requires users to instantiate a UnitRegistry instance (unrelated to the unyt.UnitRegistry class), which in turn has Unit instances as attributes. Just like with unyt and astropy.units, creating a Quantity instance requires multiplying an array or scalar by a Unit instance. Exposing the UnitRegistry directly to all users like this does force users of the library to think about which system of units they are working with, which may be beneficial in some cases, however it also means that users have a bit of extra cognitive overhead they need to deal with every time they use Pint.

In addition, the Quantity class provided by Pint is not a subclass of NumPy's ndarray. Instead, it is a wrapper around an internal ndarray buffer. This simplifies the implementation of Pint by avoiding the somewhat arcane process for creating an ndarray subclass, although the Pint Quantity class must also be careful to emulate the full NumPy ndarray API so that it can be a drop-in replacement for ndarray.

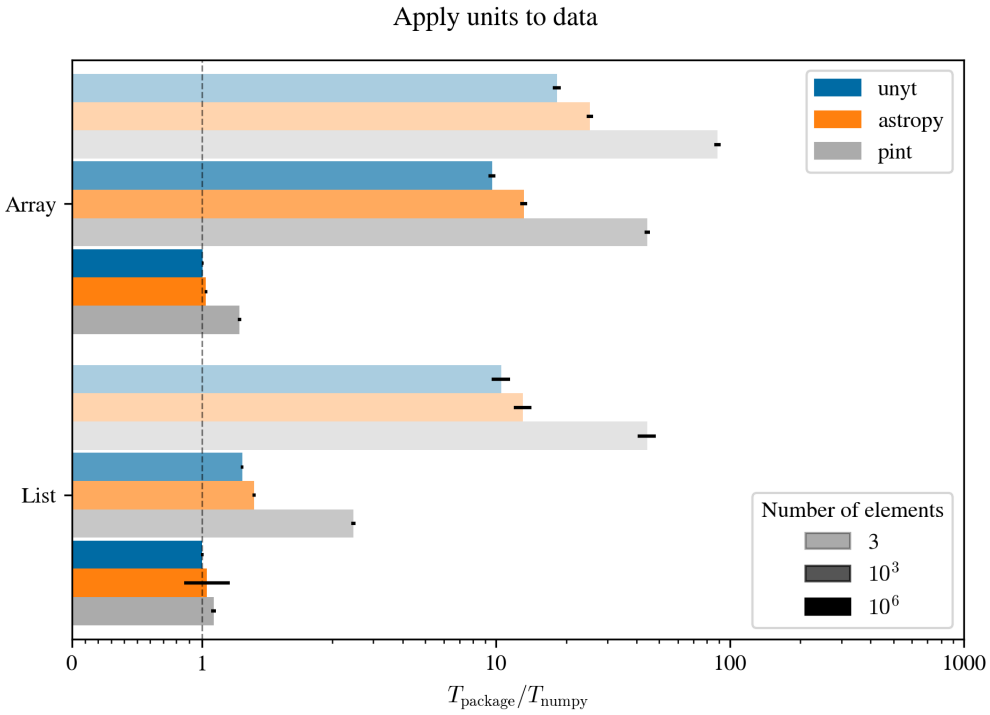

Figure 1: A benchmark comparing the ratio of the time to apply units to lists and NumPy `ndarray` instances to the time to interpret the same list or `ndarray` to an `ndarray`. This ratio, $T_{\mathrm{package}}/T_{\mathrm{numpy}}$, corresponds to the overhead of converting data to work with one of the three packages. Values close to unity correspond to zero or negligible overhead, while values larger than unity correspond to measurable overhead. Optimally all values would be near unity. In practice, applying units to small arrays incurs substantial overhead. Each test is shown for three different sizes of input data, including inputs with size 3, 1,000, and 1,000,000. The black lines at the top of the bars indicate the sample standard deviation. The T_{numpy} time is calculated by benchmarking the time to perform `np.asarray(data)` where `data` is either a `list` or an `ndarray`.

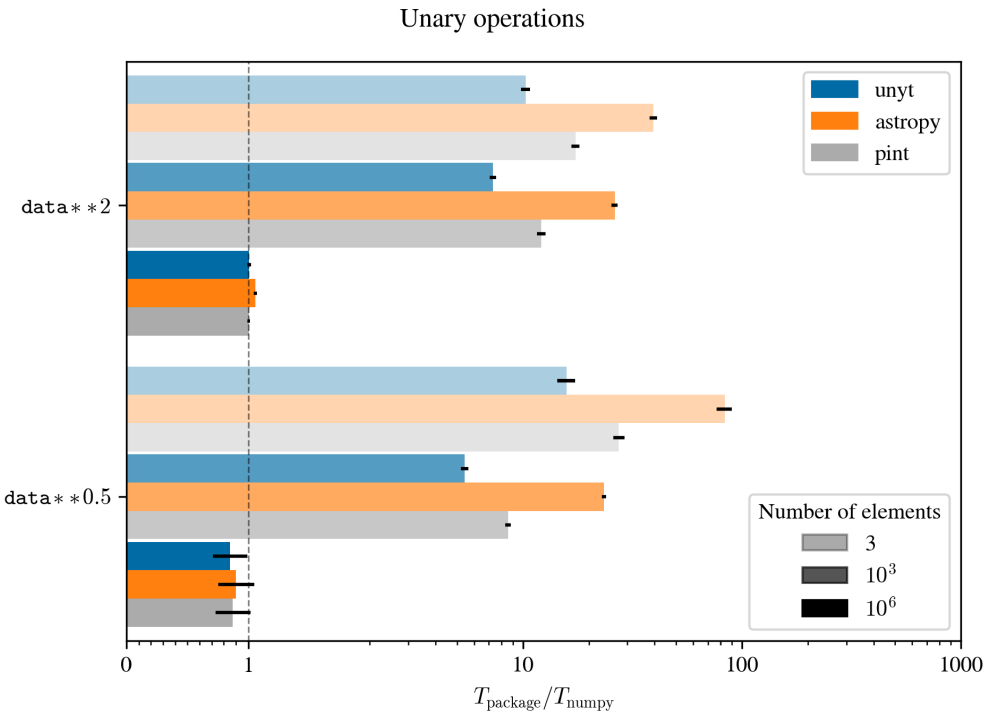

Figure 2: A benchmark comparing the time to square an array and to take the square root of an array. See Figure 1 for a detailed explanation of the plot style.

Finally, in comparing the output of our benchmarks of `Pint`, `astropy.units`, and `unyt`, we found that in-place operations making use of a NumPy `ufunc` will unexpectedly strip units in `Pint`. For example, if `a` and `b` are `Pint Quantity` instances, `np.add(a, b, out=out))` will operate on `a` and `b` as if neither have units attached. Interestingly, without the `out` keyword, `Pint` does get the correct answer, so it is possible that this is a bug in `Pint`, and we have reported it as such upstream (see https://github.com/hgrecco/pint/issues/644).

Performance Comparison

Checking units will always add some overhead over using hard-coded unit conversion factors. Thus a library that is entrusted with checking units in an application should incur the minimum possible overhead to avoid triggering performance regressions after integrating unit checking into an application. Optimally, a unit library will add zero overhead regardless of the size of the array. In practice that is not the case for any of the three libraries under consideration, and there is a minimum array size above which the overhead of doing a mathematical operation exceeds the overhead of checking units. It is thus worth benchmarking unit libraries in a fair manner, comparing with the same operation implemented using plain NumPy.

Here we present such a set of benchmarks. We made use of the `perf` (Stinner, 2018) Python benchmarking tool, which not only provides facilities for establishing the statistical significance of a benchmark run, but also can tune a linux system to turn off operating system and hardware features like CPU throttling that might introduce variance in a benchmark. We made use of a Dell Latitude E7270 laptop equipped with an Intel i5-6300U CPU clocked at 2.4 GHz. The testing environment was based on `Python`

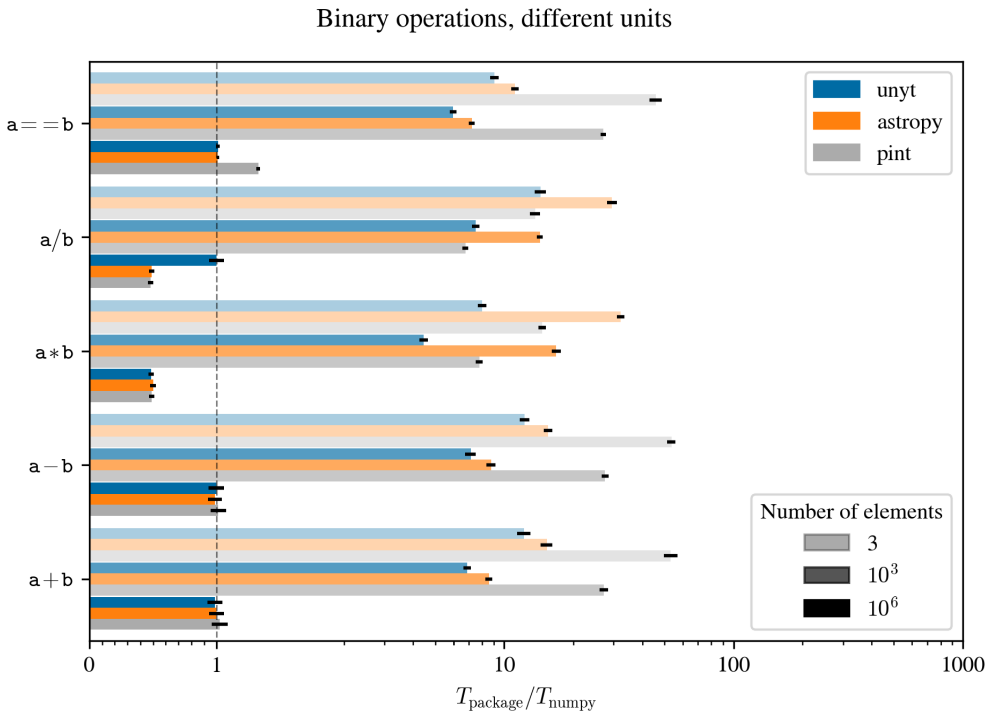

Figure 3: A benchmark comparing the time to perform various binary arithmetic operations on input operans that have different but dimensionallty compatible units. See Figure 1 for a detailed explanation of the plot style.

3.6.3 and had `NumPy 1.14.2`, `sympy 1.1.1`, `fastcache 1.0.2`, `Astropy 3.0.1`, and `Pint 0.8.1` installed. `fastcache` (Brady, 2017) is an optional dependency of `SymPy` that provides an optimized LRU cache implemented in C that can substantially speed up `SymPy`. The system was instrumented using `perf system tune` to turn off CPU features that might interfere with stable benchmarks. We did not make any boot-time Linux kernel parameter changes.

For each of the benchmarks we show the ratio of the time to perform an operation with one of `unyt`, `Pint`, and `astopy.units`, T_{package}, to the time it takes for NumPy to perform the equivalent operation, T_{numpy}. For example, for the comparison of the performance of `np.add(a, b)` where `a` and `b` have different units with the same dimension, the corresponding benchmark to generate T_{numpy} would use the code `np.add(a, c*b)` where `a` and `b` would be `ndarray` instances and `c` would be the floating point conversion factor between the units of `a` and `b`. Much of the time in T_{package} relative to T_{numpy} is spent in the respective packages calculating the appropriate conversion factor `c`. Thus the comparisons below depict very directly the overhead for using a unit library over an equivalent operation that uses hard-coded unit-conversion factors.

In some cases when the operands of an operation have different dimensionally compatible units, using a unit library will produce a result *faster* than a pure-numpy implementation. In cases where this happens, the resulting $T_{\text{package}}/T_{\text{numpy}}$ measurement will come out less than unity. As an example, consider the operation of one milligram multiplied by 3 kilograms. In this case one could write down the answer as e.g. `3000 mg`, `0.003 kg`, or `3 kg*g`. In the former two cases, one of the operands needs to be scaled by a floating point conversion factor to ensure that both operands have the same unit before the pure-numpy implementation can actually evaluate the result, since each array can only have one unit.

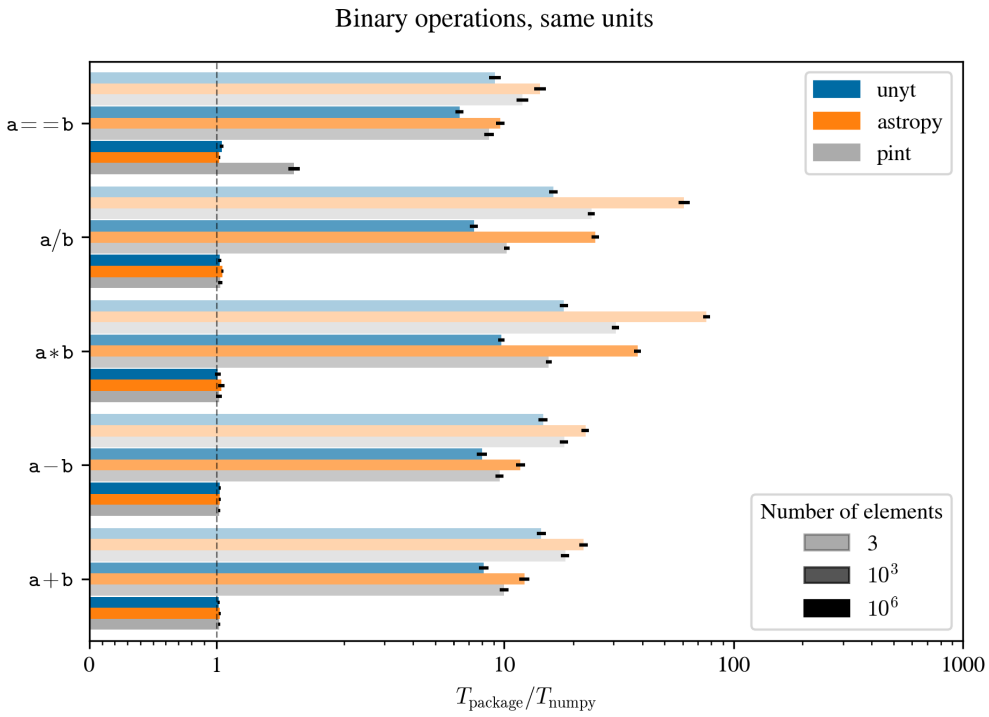

Figure 4: A benchmark comparing the time to perform various binary arithmetic operations on input operands that have the same units. See Figure 1 for a detailed explanation of the plot style.

In the latter case the conversion factor is implicitly handled by the unit metadata. Note that this effect will only happen if the operands of an operation have different units. If the units are the same there is no need to calculate a conversion factor to convert the operands to the same unit system, so the pure-numpy operation avoids the extra cost that a unit library avoids in all cases.

Applying units to data

In Figure 1 we plot the overhead for applying units to data, showing both Python lists and NumPy `ndarray` instances as the input to apply data to. Since all three libraries eventually convert input data to a NumPy `ndarray`, the comparison with array inputs more explicitly shows the overhead for *just* applying units to data. When applying units to a list, all three libraries as well as NumPy need to first copy the contents of the list into a NumPy array or a subclass of `ndarray`. This explains why the overhead is systematically lower when starting with a list.

In all cases, `unyt` either is fastest by a statistically significant margin, or ties with `astropy`. Even for large input arrays, `Pint` still has statistically significant overhead, while both `unyt` and `astropy.units` have negligible overhead once the input array size reaches 10^6 elements.

Unary arithmetic operations

Expressions involving powers of data with units, including integer and fractional powers, are very common in the physical sciences. It is therefore very important for a library that handles units to be able to track this case in a performant way. In Figure 2 we present a

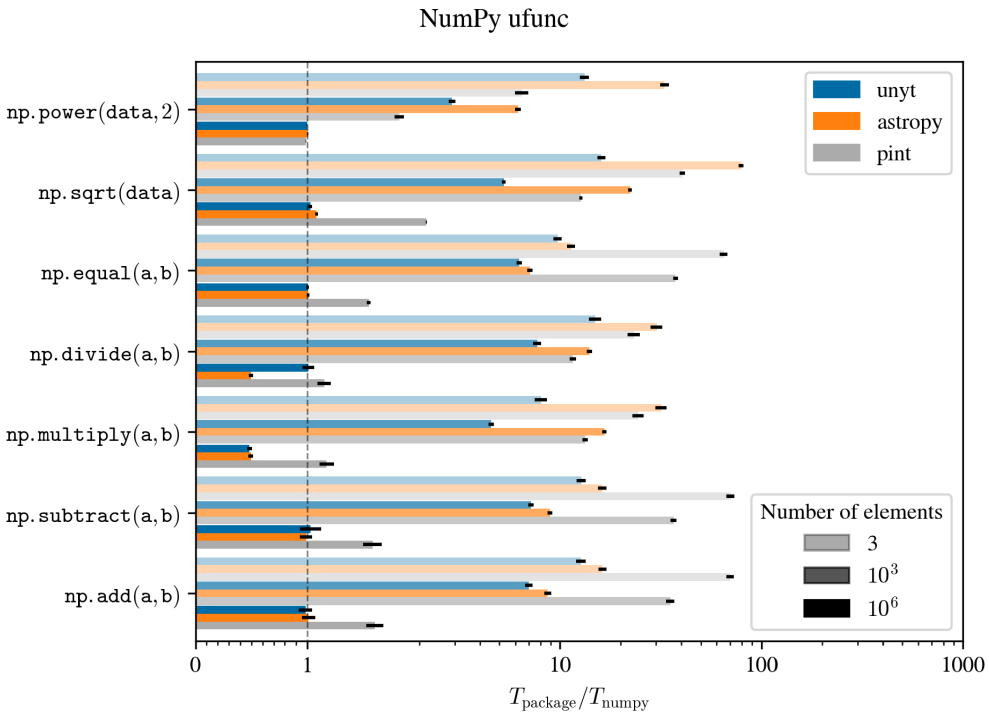

Figure 5: A benchmark comparing the overhead for computing various NumPy `ufunc` operations. The operands of all binary `ufuncs` have the same units. See Figure 1 for a detailed explanation of the plot style.

benchmark comparing `Pint`, `unyt`, and `astropy.units` for the squaring and square root operation. In all cases, `unyt` has the lowest overhead, with `Pint` coming in second, and `astropy.units` trailing. Note that the x-axis is plotted on a log scale, so `astropy` is as much as 4 times slower than `unyt` for these operations.

Binary arithmetic operations

Binary operations form the core of arithmetic. It is vital for a library that handles unit manipulation to both transparently convert units when necessary and to ensure that expressions involving quantities with units are dimensionally consistent. In Figure 3 and 4 we present benchmarks for binary arithmetic expressions, both with input data that has the same units and with input data with different units but the same dimensions. In most cases, `unyt` has less overhead than both `astropy` and `Pint`, although there are a few anomalies that are worth explaining in more detail. For comparison operations, `Pint` exhibits a slowdown even on large input arrays. This is not present for other binary operations, so it is possible that this overhead might be eliminated with a code change in `Pint`.

For multiplication on large arrays, all three libraries have measured overhead of ~ 0.5 than that of the "equivalent" Numpy operation. See the discussion in the "Performance Comparison" section above for why this happens, but briefly, in all three libraries there is no need to multiply the result of a multiplication operation by an additional constant to ensure the result has a well-defined unit, while a pure NumPy implementation would need to multiply the result by a constant to ensure both operands of the operation are in the same units.

For division, both `Pint` and `astropy.units` exhibit the same behavior as for multiplication, and for similar reasons: the result of the division operation is output with units given by the ratio of the input units. On the other hand, `unyt` will automatically cancel the dimensionally compatible units in the ratio and return a result with dimensionless units. To make that concrete, in `astropy` and `Pint`, the result of `(4*g) / (2*kg)` is `2 g/kg` while `unyt` would report `.002`.

NumPy `ufunc` performance

Lastly, in Figures 5 and 6, we present benchmarks of NumPy `ufunc` operations. A NumPy `ufunc` is a fast C implementation of a basic mathematical operation. This includes arithmetic operators as well as trigonometric and special functions. By using a `ufunc` directly, one bypasses the Python object protocol and short-circuits directly to the low-level NumPy math kernels. We show both directly using the NumPy `ufunc` operators (Figure 5) as well as using the same operators with a pre-allocated output array to benchmark in-place operations.

As for the other benchmarks, `unyt` tends to have the lowest amount of overhead, although there are some significant exceptions. For `np.power`, `Pint` has the lowest overhead, except for very large input arrays, where the overhead for all three libraries is negligible. On the other hand, for `np.sqrt`, `np.equal`, `np.add`, and `np.subtract`, `Pint` still has statistically significant overhead for large input arrays. Finally, for the in-place `ufunc` comparison, `Pint` has the lowest overhead for all operations. However, as discussed above, this is because of a bug in `Pint` which causes the library to ignore units when calling a `ufunc` with the `out` keyword set.

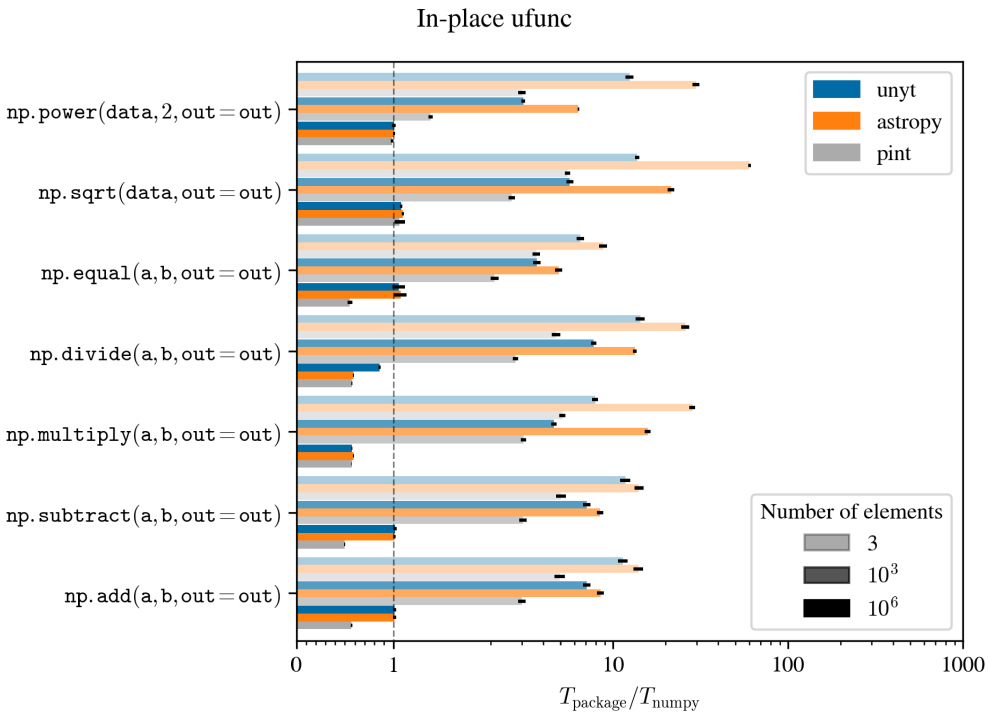

Figure 6: The same as Figure 5, but with in-place `ufunc` operations. See Figure 1 for a detailed explanation of the plot style.

Conclusions

In this paper we present the `unyt` library, giving background on the reasons for its existence and some historical context for its origin. We also present a set of benchmarks for common arithmetic operations, comparing the performance of `unyt` with `Pint` and `astropy.units`. In general, we find that `unyt` either outperforms or matches the performance `astropy.units` and `Pint`, depending on the operation and size of the input data. We also demonstrate that the `unyt` library constitutes a smaller codebase with higher test coverage than both `Pint` and `astropy.units`.

Acknowledgements

NJG would like to thank Brandon Carswell and Alex Farthing of the NCSA IT staff for providing a laptop with Linux installed for the performance benchmark. This work was supported by NSF grant OAC-1663914 (NJG, MJT), by the Gordon and Betty Moore Foundation's Data-Driven Discovery Initiative through Grant GBMF4561 (MJT) and by NASA through Einstein Postdoctoral Fellowship grant number PF7-180166 awarded by the Chandra X-ray Center, which is operated by the Smithsonian Astrophysical Observatory for NASA under contract NAS8-03060 (ALR).

References

Bekolay, T. (2013). *A comprehensive look at representing physical quantities in python.* Retrieved from https://www.youtube.com/watch?v=N-edLdxiM40

Board, M. C. O. M. I. (1999). *Mars climate orbiter mishap investigation board: Phase i report.* Jet Propulsion Laboratory. Retrieved from https://books.google.com/books?id=4OMIHQAACAAJ

Brady, P. (2017). Fastcache. *GitHub repository.* https://github.com/pbrady/fastcache; GitHub.

Collette, A. (2013). *Python and hdf5.* O'Reilly.

Croton, D. J. (2013). Damn You, Little h! (Or, Real-World Applications of the Hubble Constant Using Observed and Simulated Data). *Publications of the Astronomical Society of Australia*, *30*, e052. doi:10.1017/pasa.2013.31

Danial, A. (2018). Cloc. *GitHub repository.* https://github.com/AlDanial/cloc; GitHub.

Gopinath, R., Jensen, C., & Groce, A. (2014). Code coverage for suite evaluation by developers. In *Proceedings of the 36th international conference on software engineering*, ICSE 2014 (pp. 72–82). New York, NY, USA: ACM. doi:10.1145/2568225.2568278

Grecco, H. E. (2018). Pint. *GitHub repository.* https://github.com/hgrecco/pint; GitHub.

Koru, A. G., Zhang, D., & Liu, H. (2007). Modeling the effect of size on defect proneness for open-source software. In *Predictor models in software engineering, 2007. PROMISE'07: ICSE workshops 2007. International workshop on* (pp. 10–10). doi:10.1109/PROMISE.2007.9

Lipow, M. (1982). Number of faults per line of code. *IEEE Trans. Softw. Eng.*, *8*(4), 437–439. doi:10.1109/TSE.1982.235579

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., et al. (2017). SymPy: Symbolic computing in python. *PeerJ Computer Science*, *3*, e103. doi:10.7717/peerj-cs.103

Oliphant, T. E. (2006). *A guide to numpy.* Trelgol Publishing.

Pontzen, A., Roškar, R., Stinson, G. S., Woods, R., Reed, D. M., Coles, J., & Quinn, T. R. (2013). pynbody: Astrophysics Simulation Analysis for Python. http://ascl.net/1305.002.

Stark, C. W. (2012). Dimensionful. *GitHub repository.* https://github.com/caseywstark/dimensionful; GitHub.

Stinner, V. (2018). Perf. *GitHub repository.* https://github.com/vstinner/perf; GitHub.

The Astropy Collaboration, Price-Whelan, A. M., Sipőcz, B. M., Günther, H. M., Lim, P. L., Crawford, S. M., Conseil, S., et al. (2018). The Astropy Project: Building an inclusive, open-science project and status of the v2.0 core package. *ArXiv e-prints.*

Turk, M. J., Smith, B. D., Oishi, J. S., Skory, S., Skillman, S. W., Abel, T., & Norman, M. L. (2011). yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data. *The Astrophysical Journal Supplement Series*, *192*, 9. doi:10.1088/0067-0049/192/1/9

Walt, S. van der, Colbert, S. C., & Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, *13*(2), 22–30. doi:10.1109/MCSE.2011.37