The Journal of Open Source Software

AutoUncertainties: A Python Package for Uncertainty Propagation

Varchas Gopalaswamy [1*] **and Ethan Mentzer** [1*]

1 Laboratory for Laser Energetics, Rochester, USA * These authors contributed equally.

DOI: 10.21105/joss.08037

Software

- Review
- Repository
- Archive

Editor: Øystein Sørensen

Reviewers:

- @pescap
- @JakobBD
- @damar-wicaksono

Submitted: 21 March 2025
Published: 01 July 2025

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

Summary

Propagation of uncertainties is of great utility in the experimental sciences. While the rules of (linear) uncertainty propagation are straightforward, managing many variables with uncertainty information can quickly become complicated in large scientific software stacks. Often, this requires programmers to keep track of many variables and implement custom error propagation rules for each mathematical operator and function. The Python package `AutoUncertainties`, described here, provides a solution to this problem.

Statement of Need

`AutoUncertainties` is a Python package for uncertainty propagation of independent random variables. It provides a drop-in mechanism to add uncertainty information to Python scalar and `NumPy` (Harris et al., 2020) array objects. It implements manual propagation rules for the Python dunder math methods, and uses automatic differentiation via `JAX` (Bradbury et al., 2018) to propagate uncertainties for most `NumPy` methods applied to both scalar and `NumPy` array variables. In doing so, it eliminates the need for carrying around additional uncertainty variables or for implementing custom propagation rules for any `NumPy` operator with a gradient rule implemented by `JAX`. Furthermore, in most cases, it requires minimal modification to existing code—typically only when uncertainties are attached to central values.

Prior Work

To the author's knowledge, the only existing error propagation library in Python is the `uncertainties` (Lebigot et al., 2024) package, which inspired the current work. While extremely useful, the `uncertainties` package relies on hand-implemented rules and functions for uncertainty propagation of array and scalar data. This is mostly trivial for Python's intrinsic arithmetic and logical operations such as `__add__`, however it becomes problematic for more advanced mathematical operations. For instance, calculating the uncertainty propagation due to the cosine function requires the import of separate math libraries, rather than being able to use `NumPy` directly.

```
# Using uncertainties v3.2.3
import numpy as np
from uncertainties import unumpy, ufloat
arr = np.array([ufloat(1, 0.1), ufloat(2, 0.002)])
unumpy.cos(arr)  # calculation succeeds
np.cos(arr)      # raises an exception
```

The example above illustrates a primary limitation of the `uncertainties` package: arrays of `ufloat` objects cannot be seamlessly integrated with common `NumPy` functions, and can only

be operated on by the unumpy suite of functions.

Implementation

For a function $f : \mathbb{R}^n \to \mathbb{R}^m$ of n independent variables, linear uncertainty propagation can be computed via the simple rule

$$\delta f_j(\mathbf{x})^2 = \sum_i^n \left(\frac{\partial f_j}{\partial x_i} \delta x_i \right)^2, \qquad j \in [1, m].$$

To compute $\frac{\partial f_j}{\partial x_i}$ for arbitrary f, the implementation in AutoUncertainties relies on automatic differentiation provided by JAX. Calls to any NumPy array function or universal function (ufunc) are intercepted via the __array_function__ and __array_ufunc__ mechanism, and dispatched to a NumPy wrapper routine that computes the Jacobian matrix via jax.jacfwd.

The user API for the Uncertainty object exposes a number of properties and methods, of which some of the most important are:

- value -> float: The central value of the object.
- error -> float: The error (standard deviation) of the object.
- relative -> float: The relative error (i.e. error / value) of the object.
- plus_minus(self, err: float) -> Uncertainty: Adds error (in quadrature).

These attributes and methods can be used in the following manner:

```
from auto_uncertainties import Uncertainty
u1 = Uncertainty(5.25, 0.75)
u2 = Uncertainty(1.85, 0.4)

print(u1)                 # 5.25 +/- 0.75
print(u1.value)           # 5.25
print(u1.error)           # 0.75
print(u1.relative)        # 0.142857
print(u1.plus_minus(0.5)) # 5.25 +/- 0.901388

# Construct a vector Uncertainty from a sequence.
seq = Uncertainty([u1, u2])
print(seq.value)  # [5.25 1.85]
print(seq.error)  # [0.75 0.4 ]
```

Of course, one of the most important aspects of AutoUncertainties is its seamless support for NumPy:

```
import numpy as np
from auto_uncertainties import Uncertainty
vals = np.array([0.5, 0.75])
errs = np.array([0.05, 0.3])
u = Uncertainty(vals, errs)

print(np.cos(u))  # [0.877583 +/- 0.0239713, 0.731689 +/- 0.204492]
```

This is in contrast to the uncertainties package, which would have necessitated using the unumpy module of hand-implemented NumPy function analogs.

The Uncertainty class automatically determines which methods should be implemented based on whether it represents a vector uncertainty, or a scalar uncertainty. When instantiated with sequences or NumPy arrays, vector-based operations are enabled; when instantiated with scalars, only scalar operations are permitted.

AutoUncertainties also provides certain exceptions, helper functions, accessors, and display rounding adjustors, whose details can be found in the documentation.

Support for Pint

AutoUncertainties provides some support for working with objects from the Pint package (Grecco & Chéron, 2025). For example, Uncertainty objects can be instantiated from pint.Quantity objects, and then automatically wrapped into new pint.Quantity objects via the from_quantities method. This guarantees that unit information is preserved when moving between Uncertainty objects and pint.Quantity objects.

```
from auto_uncertainties import Uncertainty
from pint import Quantity

val = Quantity(2.24, 'kg')
err = Quantity(0.208, 'kg')
new_quantity = Uncertainty.from_quantities(val, err)

print(new_quantity)        # 2.24 +/- 0.208 kilogram
print(type(new_quantity))  # <class 'pint.Quantity'>
```

Current Limitations and Future Work

Dependent Random Variables

To simplify operations on Uncertainty objects, AutoUncertainties assumes all variables are independent. This means that, in the case where the programmer assumes dependence between two or more Uncertainty objects, unexpected and counter-intuitive behavior may arise during uncertainty propagation. This is a common pitfall when working with Uncertainty objects, especially since the package will not prevent programmers from manipulating variables in a manner that implies dependence. Examples of this behavior, along with certain potential workarounds, can be found here in the documentation. In general, most binary operations involving the same variable twice will produce undesired results (for instance, performing X - X, where X is an Uncertainty object, will *not* result in a standard deviation of zero).

The workarounds are nevertheless cumbersome, and cause AutoUncertainties to fall somewhat short of the original goals of automated error propagation. In principle, this could be addressed by storing a full computational graph of the result of chained operations, similar to what is done in uncertainties. However, the complexity of such a system places it out of scope for AutoUncertainties at this time.

Further Information

Full API information and additional usage examples can be found on the documentation website. All source code for the project is stored and maintained on the AutoUncertainties GitHub repository, where contributions, suggestions, and bug reports are welcome.

Acknowledgements

This material is based upon work supported by the Department of Energy [National Nuclear Security Administration] University of Rochester "National Inertial Confinement Fusion Program" under Award Number(s) DE-NA0004144, and Department of Energy [Office of Fusion Energy Sciences] University of Rochester "Applications of Machine Learning and Data Science

to predict, design and improve laser-fusion implosions for inertial fusion energy" under Award Number(s) DE-SC0024381.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

References

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.5.1). https://github.com/jax-ml/jax

Grecco, H. E., & Chéron, J. (2025). Pint: Makes units easy. In *GitHub repository*. GitHub. https://github.com/hgrecco/pint

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Lebigot, E. O., Newville, M., Deil, C., Davar, G., Shanks, W., Savara, A., Yurchak, R., andrewgsavage, ep12, Gerber, J., Pieters, M., Romano, P., Riley, W., Baldwin, A., Abel, B., Burr, C., op3, Eendebak, P., & Cladé, P. (2024). *Lmfit/uncertainties: 3.2.2* (Version 3.2.2). Zenodo. https://doi.org/10.5281/zenodo.12682754