uncertainties Python package Documentation

Release 3.1.7

Eric O. LEBIGOT (EOL)

Apr 30, 2024

CONTENTS

The uncertainties package is an open source Python library for doing calculations on numbers that have uncertainties (like 3.14±0.01) that are common in many scientific fields. The calcualtions done with this package will propagate the uncertainties to the result of mathematical calculations. The *uncertainties* package takes the pain and complexity out of uncertainty calculations and error propagation. Here is a quick taste of how to use *uncertainties*:

```
>>> from uncertainties import ufloat
>>> x = ufloat(2, 0.1)   # x = 2+/-0.1
>>> y = ufloat(3, 0.2)   # y = 3+/-0.2
>>> print(2*x)
4.00+/-0.20
>>> print(x+y)
5.00+/-0.22
>>> print(x*y)
6.0+/-0.5
```

The *uncertainties* library calculates uncertainties using linear error propagation theory by automatically *calculating derivatives* and analytically propagating these to the results. Correlations between variables are automatically handled. This library can also yield the derivatives of any expression with respect to the variables that have uncertain values. For other approaches, see soerp (using higher-order terms) and mcerp (using a Monte-Carlo approach).

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Installation and Credits

1.1.1 Download and Installation

The *uncertainties* package supports Python versions 3.8 and higher. Earlier versions of Python are not tested, but may still work. Development version of Python (currently, 3.13) are likely to work, but are not regularly tested.

To install *uncertainties*, use:

```
pip install uncertainties
```

You can upgrade from an older version of *uncertainties* with:

```
pip install --upgrade uncertainties
```

Other packaging systems such as Anaconda, MacPorts, or Linux package managrer may also maintain packages for *uncertainties*, so that you may also be able to install using something like

```
conda install -c conda-forge uncertainties
```

```
sudo port install py**-uncertainties
```

or

```
sudo apt get python-uncertainties
```

depending on your platform and installation of Python. For all installations of Python, using *pip* should work and is therefore recommended.

1.1.2 Souce code and Development Version

You can download the latest source package archive from the Python Package Index (PyPI) and unpack it, or from the GitHub releases page. This package can be unpacked using *unzip*, *tar xf* , or other similar utilites, and then installeed with

```
python -m pip install .
```

To work with the development version, use *git* to fork or clone the code:

```
git clone git@github.com:lmfit/uncertainties.git
```

The *uncertainties* package is written in pure Python and has no external dependencies. If available (and recommended), the NumPy package can be used. Running the test suite requires *pytest* and *pytest_cov*, and building these docs requires *sphinx*. To install these optional packages, use one of:

```
pip install ".[arrays]"    # to install numpy
pip install ".[test]"      # to enable running the tests
pip install ".[doc]"       # to enable building the docs
pip install ".[all]"       # to enable all of these options
```

1.1.3 Getting Help

If you have questions about *uncertainties* or run into trouble, use the GitHub Discussions page. For bug reports, use the GitHub Issues pages.

1.1.4 Credits

The *uncertainties* package was written and developed by Eric O. LEBIGOT (EOL). EOL also maintained the package until 2024, when the GitHub project was moved to the lmfit GitHub organization to allow more sustainable development and maintenance. Currentt members of the devlopment and maintenance team include Andrew G Savage, Justin Gerber, Eric O Legibot, Matt Newville, and Will Shanks. Contributions and suggestions for development are welcome.

1.1.5 How to cite this package

If you use this package for a publication, please cite it as *Uncertainties: a Python package for calculations with uncertainties*, Eric O. LEBIGOT. A version number can be added, but is optional.

1.1.6 Acknowledgments

Eric O. LEBIGOT (EOL) thanks all the people who made generous donations: that help to keep this project alive by providing positive feedback.

EOL greatly appreciates having gotten key technical input from Arnaud Delobelle, Pierre Cladé, and Sebastian Walter. Patches by Pierre Cladé, Tim Head, José Sabater Montes, Martijn Pieters, Ram Rachum, Christoph Deil, Gabi Davar, Roman Yurchak and Paul Romano are gratefully acknowledged.

EOL also thanks users who contributed with feedback and suggestions, which greatly helped improve this program: Joaquin Abian, Jason Moore, Martin Lutz, Víctor Terrón, Matt Newville, Matthew Peel, Don Peterson, Mika Pflueger, Albert Puig, Abraham Lee, Arian Sanusi, Martin Laloux, Jonathan Whitmore, Federico Vaggi, Marco A. Ferra, Hernan Grecco, David Zwicker, James Hester, Andrew Nelson, and many others.

EOL is grateful to the Anaconda, macOS and Linux distribution maintainers of this package (Jonathan Stickel, David Paleino, Federico Ceratto, Roberto Colistete Jr, Filipe Pires Alvarenga Fernandes, and Felix Yan) and also to Gabi Davar and Pierre Raybaut for including it in Python(x,y) and to Christoph Gohlke for including it in his Base distribution of scientific Python packages for Windows.

1.1.7 License

1.2 User Guide

1.2.1 Basic usage

Basic mathematical operations involving numbers with uncertainties requires importing the `ufloat()` function which creates a `Variable`: numbet with both a nominal value and an uncertainty.

```
>>> from uncertainties import ufloat
>>> x = ufloat(2.7, 0.01)   #  a Variable with a value 2.7+/-0.01
```

The *uncertainties* module contains sub-modules for *advanced mathematical functions*, and *arrays and matrices*, which can be accessed with:

```
>>> import uncertainties
```

1.2.2 Creating Variables: numbers with uncertainties

To create a number with uncertainties or *Variable*, use the `ufloat()` function, which takes a *nominal value* ((which can be interpreted as the most likely value, or the mean or central value of the distribution of values), a *standard error* ((the standard deviation or $1-\sigma$ uncertainty), and an optional *tag*:

```
>>> x = ufloat(2.7, 0.01)  # x = 2.7+/-0.01
>>> y = ufloat(4.5,  1.2, tag='y_variable')  # x = 4..5+/-1.2
```

You can access the nominal value and standard deviation for any Variable with the *nominal_value* and *std_dev* attributes:

```
>>> print(x.nominal_value,  x.std_dev)
2.7 0.01
```

uncertainties Variables can also be created from one of many string representations. The following forms will all create Variables with the same value:

```
>>> from uncertainties import ufloat_fromstr
>>> x = ufloat(0.2, 0.01)
>>> x = ufloat_fromstr("0.20+/-0.01")
>>> x = ufloat_fromstr("(2+/-0.1)e-01")  # Factored exponent
>>> x = ufloat_fromstr("0.20(1)")  # Short-hand notation
>>> x = ufloat_fromstr("20(1)e-2")  # Exponent notation
>>> x = ufloat_fromstr(u"0.20±0.01")  # Pretty-print form
>>> x = ufloat_fromstr("0.20")  # Automatic uncertainty of +/-1 on last digit
```

ufloat(), ufloat_fromstr(), and Variable

The most common and important functions for creating uncertain Variables are *ufloat()* and *ufloat_fromstr()*.

uncertainties.**ufloat**(*nominal_value*, *std_dev=None*, *tag=None*)

Create an uncertainties Variable

Arguments:

nominal_value: float
nominal value of Variable

std_dev: float or *None*
standard error of Variable, or *None* if not available [*None*]

tag: string or *None*
optional tag for tracing and organizing Variables ['None']

Returns:

uncertainties Variable

Examples

```
>>> a = ufloat(5, 0.2)
>>> b = ufloat(1000, 30, tag='kilo')
```

Notes:

1. *nominal_value* is typically interpreted as *mean* or *central value*
2. ***std_dev* is typically interpreted as *standard deviation* or the** 1-sigma level uncertainty.
3. The returned Variable will have attributes *nominal_value*, *std_dev*, and *tag* which match the input values.

uncertainties.**ufloat_fromstr**(*representation*, *tag=None*)

Create an uncertainties Variable from a string representation. Several representation formats are supported.

Arguments:

representation: string
string representation of a value with uncertainty

tag: string or *None*
optional tag for tracing and organizing Variables ['None']

Returns:

uncertainties Variable.

Notes:

1. Invalid representations raise a ValueError.
2. Using the form "nominal(std)" where "std" is an integer creates a Variable with "std" giving the least significant digit(s). That is, "1.25(3)" is the same as *ufloat(1.25, 0.03)*, while "1.25(3.)" is the same as *ufloat(1.25, 3.)*

Examples:

```
>>> x = ufloat_fromsstr("12.58+/-0.23")  # = ufloat(12.58, 0.23)
>>> x = ufloat_fromsstr("12.58 ± 0.23")  # = ufloat(12.58, 0.23)
>>> x = ufloat_fromsstr("3.85e5 +/- 2.3e4")  # = ufloat(3.8e5, 2.3e4)
>>> x = ufloat_fromsstr("(38.5 +/- 2.3)e4")  # = ufloat(3.8e5, 2.3e4)
```

```
>>> x = ufloat_fromsstr("72.1(2.2)")  # = ufloat(72.1, 2.2)
>>> x = ufloat_fromsstr("72.15(4)")  # = ufloat(72.15, 0.04)
>>> x = ufloat_fromstr("680(41)e-3")  # = ufloat(0.68, 0.041)
>>> x = ufloat_fromstr("23.2")  # = ufloat(23.2, 0.1)
>>> x = ufloat_fromstr("23.29")  # = ufloat(23.29, 0.01)
```

```
>>> x = ufloat_fromstr("680.3(nan)") # = ufloat(680.3, numpy.nan)
```

class uncertainties.**Variable**(*value*, *std_dev*, *tag=None*)

Representation of a float-like scalar Variable with its uncertainty.

Variablees are independent from each other, but correlations between them are handled through the AffineScalarFunc class.

1.2.3 Basic math with uncertain Variables

:class:`Variable`s can be used in basic mathematical calculations ('+', '-', '*', '/', '**') as with other Python numbers and variables.

```
>>> t = ufloat(0.2, 0.01)
>>> double = 2.0*t
>>> print(double)
0.4+/-0.02
>>> square = t**2
>>> print(square)
0.040+/-0.004
```

When adding two Variables, the uncertainty in the result is the quadrature sum (square-root of the sum of squares) of the uncertaities of the two Variables:

```
>>> x = ufloat(20, 4)
>>> y = ufloat(12, 3)
>>> print(x+y)
32.0+/-5.0
```

We can check that error propagation when adding two independent variables:

```
>>> from math import sqrt
>>> (x+y).std_dev == sqrt(x.std_dev**2 + y.std_dev**2)
True
```

Multiplying two Variables will properly propagate those uncertainties too:

```
>>> print(x*y)
240.0+/-76.83749084919418
>>> (x*y).std_dev == (x*y).nominal_value *  sqrt((x.std_dev/x.nominal_value)**2 + (y.std_
↪dev/y.nominal_value)**2 )
True
```

But note that adding a Variable to itself does not add its uncertaintes in quadrature, but are simply scaled:

```
>>> print(x+x)
40.0+/-8.0
>>> print(3*x + 10)
70.0+/-12.0
```

It is important to understand that calculations done with Variable know about the correlation between the Variables. Variables created with *ufloat()* (and *ufloat_fromstr()*) are completely uncorrelated with each other, but are known to be completely correlated with themselves. This means that

```
>>> x = ufloat(5, 0.5)
>>> y = ufloat(5, 0.5)
>>> x - y
0.0+/-0.7071067811865476
>>> x - x
0.0+/-0
```

For two *different* Variables, uncorrelated uncertainties will be propagated. But when doing a calculation with a single Variable, the uncertainties are correlated, and calculations will reflect that.

1.2.4 Mathematical operations with uncertain Variables

Besides being able to apply basic mathematical operations to uncertaintties Variables, this package provides generalized versions of 40 of the the functions from the standard `math` *module*. These mathematical functions are found in the `uncertainties.umath` module:

```
>>> from uncertainties.umath import sin, exp, sqrt
>>> x   = ufloat(0.2, 0.01)
>>> sin(x)
0.19866933079506122+/-0.009800665778412416
>>> sin(x*x)
0.03998933418663417+/-0.003996800426643912
>>> exp(-x/3.0)
```

(continues on next page)

(continued from previous page)

```
0.9355069850316178+/-0.003118356616772059
>>> sqrt(230*x + 3)
7.0+/-0.16428571428571428
```

The functions in the `uncertainties.umath` module include:

> 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc'

1.2.5 Comparison operators

Comparison operators ('==', '!=', '>', '<', '>=', and '<=') for Variables with uncertainties are somewhat complicated, and need special atention. As we hinted at above, and will explore in more detai below and in the *Technical Guide*, this relates to the correlation between Variables.

Equality and inequality comparisons

If we compare the equality of two Variables with the same nominal value and uncertainty, we see

```
>>> x = ufloat(5, 0.5)
>>> y = ufloat(5, 0.5)
>>> x == x
True
>>> x == y
False
```

The difference here is that although the two Python objects have the same nominal value and uncertainty, these are indepedent, uncorrelated values. It is not exactly true that the difference is based on identity, note that

```
>>> x == (1.0*x)
True
>>> x is (1.0*x)
False
```

In order for the resuls of two calculations with uncertainties to be considered equal, the *uncertainties* package does not test whether the nomimal value and the uncertainty have the same value. Instead it checks whether the difference of the two calculations has a nominal value of 0 *and* an uncertainty of 0.

```
>>> (x -x)
0.0+/-0
>>> (x -y)
0.0+/-0.7071067811865476
```

Comparisons of magnitude

The concept of comparing the magnitude of values with uncertainties is a bit complicated. That is, a Variable with a value of 25 +/- 10 might be greater than a Varitable with a value of 24 +/- 8 most of the time, but *sometimes* it might be less than it. The *uncertainties* package takes the simple approach of comparing. That is

```
>>> a = ufloat(25, 10)
>>> b = ufloat(24, 8)
>>> a > b
True
```

Note that cobining this comparison and the above discussion of == and != can lead to a result that maybe somewhat surpring:

```
>>> a = ufloat(25, 10)
>>> b = ufloat(25, 8)
>>> a >= b
False
>>> a > b
False
>>> a == b
False
>>> a.nominal_value >= b.nominal_value
True
```

That is, since *a* is neither greeater than *b* (nomal value only) nor equal to *b*, it cannot be greater than or equal to *b*.

1.2.6 Handling NaNs and infinities

NaN values can appear in either the nominal value or uncertainty of a Variable. As is always the case, care must be exercised when handling NaN values.

While `math.isnan()` and `numpy.isnan()` will raise *TypeError* exceptions for uncertainties Variables (because an uncertainties Variable is not a float), the function `umath.isnan()` will return whether the nominal value of a Variable is NaN. Similarly, `umath.isinf()` will return whether the nominal value of a Variable is infinite.

To check whether the uncertainty is NaN or Inf, use one of `math.isnan()`, `math.isinf()`, `nupmy.isnan()`, or , `nupmy.isinf()` on the `std_dev` attribute.

1.2.7 Automatic correlations

Correlations between variables are **automatically handled** whatever the number of variables involved, and whatever the complexity of the calculation. For example, when `x` is the number with uncertainty defined above,

```
>>> square = x**2
>>> print(square)
0.040+/-0.004
>>> square - x*x
0.0+/-0
>>> y = x*x + 1
>>> y - square
1.0+/-0
```

The last two printed results above have a zero uncertainty despite the fact that `x`, `y` and `square` have a non-zero uncertainty: the calculated functions give the same value for all samples of the random variable `x`.

Thanks to the automatic correlation handling, calculations can be performed in as many steps as necessary, exactly as with simple floats. When various quantities are combined through mathematical operations, the result is calculated by taking into account all the correlations between the quantities involved. All of this is done transparently.

1.2.8 Access to the individual sources of uncertainty

The various contributions to an uncertainty can be obtained through the `error_components()` method, which maps the **independent variables a quantity depends on** to their **contribution to the total uncertainty**. According to *linear error propagation theory* (which is the method followed by *uncertainties*), the sum of the squares of these contributions is the squared uncertainty.

The individual contributions to the uncertainty are more easily usable when the variables are **tagged**:

```
>>> u = ufloat(1, 0.1, "u variable")  # Tag
>>> v = ufloat(10, 0.1, "v variable")
>>> sum_value = u+2*v
>>> sum_value
21.0+/-0.223606797749979
>>> for (var, error) in sum_value.error_components().items():
...     print("{}: {}".format(var.tag, error))
...
u variable: 0.1
v variable: 0.2
```

The variance (i.e. squared uncertainty) of the result (`sum_value`) is the quadratic sum of these independent uncertainties, as it should be (`0.1**2 + 0.2**2`).

The tags *do not have to be distinct*. For instance, *multiple* random variables can be tagged as `"systematic"`, and their contribution to the total uncertainty of `result` can simply be obtained as:

```
>>> syst_error = math.sqrt(sum(  # Error from *all* systematic errors
...     error**2
...     for (var, error) in result.error_components().items()
...     if var.tag == "systematic"))
```

The remaining contribution to the uncertainty is:

```
>>> other_error = math.sqrt(result.std_dev**2 - syst_error**2)
```

The variance of `result` is in fact simply the quadratic sum of these two errors, since the variables from `result.error_components()` are independent.

1.2.9 Covariance and correlation matrices

Covariance matrix

The covariance matrix between various variables or calculated quantities can be simply obtained:

```
>>> sum_value = u+2*v
>>> cov_matrix = uncertainties.covariance_matrix([u, v, sum_value])
```

has value

```
[[0.01, 0.0,  0.01],
 [0.0,  0.01, 0.02],
 [0.01, 0.02, 0.05]]
```

In this matrix, the zero covariances indicate that `u` and `v` are independent from each other; the last column shows that `sum_value` does depend on these variables. The *uncertainties* package keeps track at all times of all correlations between quantities (variables and functions):

```
>>> sum_value - (u+2*v)
0.0+/-0
```

Correlation matrix

If the NumPy package is available, the correlation matrix can be obtained as well:

```
>>> corr_matrix = uncertainties.correlation_matrix([u, v, sum_value])
>>> corr_matrix
array([[ 1.        ,  0.        ,  0.4472136 ],
       [ 0.        ,  1.        ,  0.89442719],
       [ 0.4472136 ,  0.89442719,  1.        ]])
```

1.2.10 Correlated variables

Reciprocally, **correlated variables can be created** transparently, provided that the NumPy package is available.

Use of a covariance matrix

Correlated variables can be obtained through the *covariance* matrix:

```
>>> (u2, v2, sum2) = uncertainties.correlated_values([1, 10, 21], cov_matrix)
```

creates three new variables with the listed nominal values, and the given covariance matrix:

```
>>> sum_value
21.0+/-0.223606797749979
>>> sum2
21.0+/-0.223606797749979
>>> sum2 - (u2+2*v2)
0.0+/-3.83371856862256e-09
```

The theoretical value of the last expression is exactly zero, like for `sum - (u+2*v)`, but numerical errors yield a small uncertainty (3e-9 is indeed very small compared to the uncertainty on `sum2`: correlations should in fact cancel the uncertainty on `sum2`).

The covariance matrix is the desired one:

```
>>> uncertainties.covariance_matrix([u2, v2, sum2])
```

reproduces the original covariance matrix `cov_matrix` (up to rounding errors).

Use of a correlation matrix

Alternatively, correlated values can be defined through:

- a sequence of nominal values and standard deviations, and
- a *correlation* matrix between each variable of this sequence (the correlation matrix is the covariance matrix normalized with individual standard deviations; it has ones on its diagonal)—in the form of a NumPy array-like object, e.g. a list of lists, or a NumPy array.

Example:

```
>>> (u3, v3, sum3) = uncertainties.correlated_values_norm(
...     [(1, 0.1), (10, 0.1), (21, 0.22360679774997899)], corr_matrix)
>>> print(u3)
1.00+/-0.10
```

The three returned numbers with uncertainties have the correct uncertainties and correlations (`corr_matrix` can be recovered through `correlation_matrix()`).

1.2.11 Making custom functions accept numbers with uncertainties

This package allows **code which is not meant to be used with numbers with uncertainties to handle them anyway**. This is for instance useful when calling external functions (which are out of the user's control), including functions written in C or Fortran. Similarly, **functions that do not have a simple analytical form** can be automatically wrapped so as to also work with arguments that contain uncertainties.

It is thus possible to take a function `f()` *that returns a single float*, and to automatically generalize it so that it also works with numbers with uncertainties:

```
>>> wrapped_f = uncertainties.wrap(f)
```

The new function `wrapped_f()` (optionally) *accepts a number with uncertainty* in place of any float *argument* of `f()` (note that floats contained instead *inside* arguments of `f()`, like in a list or a NumPy array, *cannot* be replaced by numbers with uncertainties). `wrapped_f()` returns the same values as `f()`, but with uncertainties.

With a simple wrapping call like above, uncertainties in the function result are automatically calculated numerically. **Analytical uncertainty calculations can be performed** if derivatives are provided to `wrap()`.

More details are available in the documentation string of `wrap()` (accessible through the `pydoc` command, or Python's `help()` shell function).

1.2.12 Miscellaneous utilities

It is sometimes useful to modify the error on certain parameters so as to study its impact on a final result. With this package, the **uncertainty of a variable can be changed** on the fly:

```
>>> sum_value = u+2*v
>>> sum_value
21.0+/-0.223606797749979
>>> prev_uncert = u.std_dev
>>> u.std_dev = 10
>>> sum_value
21.0+/-10.00199980003999
>>> u.std_dev = prev_uncert
```

The relevant concept is that `sum_value` does depend on the variables `u` and `v`: the *uncertainties* package keeps track of this fact, as detailed in the *Technical Guide*, and uncertainties can thus be updated at any time.

When manipulating ensembles of numbers, *some* of which contain uncertainties while others are simple floats, it can be useful to access the **nominal value and uncertainty of all numbers in a uniform manner**. This is what the `nominal_value()` and `std_dev()` functions do:

```
>>> print(uncertainties.nominal_value(x))
0.2
>>> print(uncertainties.std_dev(x))
0.01
>>> uncertainties.nominal_value(3)
3
>>> uncertainties.std_dev(3)
0.0
```

Finally, a utility method is provided that directly yields the standard score (number of standard deviations) between a number and a result with uncertainty: with `x` equal to 0.20 ± 0.01,

```
>>> x.std_score(0.17)
-3.0
```

1.2.13 Derivatives

Since the application of *linear error propagation theory* involves the calculation of **derivatives**, this package automatically performs such calculations; users can thus easily get the derivative of an expression with respect to any of its variables:

```
>>> u = ufloat(1, 0.1)
>>> v = ufloat(10, 0.1)
>>> sum_value = u+2*v
>>> sum_value.derivatives[u]
1.0
>>> sum_value.derivatives[v]
2.0
```

These values are obtained with a *fast differentiation algorithm*.

1.2.14 Additional information

The capabilities of the `uncertainties` package in terms of array handling are detailed in *Uncertainties and numpy arrays*.

Details about the theory behind this package and implementation information are given in the *Technical Guide*.

1.3 Uncertainties and numpy arrays

1.3.1 Arrays of uncertainties Variables

It is possible to put uncertainties Variable in NumPy arrays and matrices:

```
>>> arr = numpy.array([ufloat(1, 0.01), ufloat(2, 0.1)])
>>> 2*arr
[2.0+/-0.02 4.0+/-0.2]
>>> print arr.sum()
3.00+/-0.10
```

Many common operations on NumPy arrays can be performed transparently even when these arrays contain numbers with uncertainties.

1.3.2 The unumpy package

While *basic operations on arrays* that contain numbers with uncertainties can be performed without it, the `unumpy` package is useful for more advanced uses.

This package contains:

1. utilities that help with the **creation and manipulation** of NumPy arrays and matrices of numbers with uncertainties;

2. **generalizations** of multiple NumPy functions so that they also work with arrays that contain numbers with uncertainties.

Operations on arrays (including their cosine, etc.) can thus be performed transparently.

These features can be made available with

```
>>> from uncertainties import unumpy
```

Creation and manipulation of arrays and matrices

Arrays

Arrays of numbers with uncertainties can be built from values and uncertainties:

```
>>> arr = unumpy.uarray([1, 2], [0.01, 0.002])
>>> print arr
[1.0+/-0.01 2.0+/-0.002]
```

NumPy arrays of numbers with uncertainties can also be built directly through NumPy, thanks to NumPy's support of arrays of arbitrary objects:

```
>>> arr = numpy.array([ufloat(1, 0.1), ufloat(2, 0.002)])
```

Matrices

Matrices of numbers with uncertainties are best created in one of two ways. The first way is similar to using `uarray()`:

```
>>> mat = unumpy.umatrix([1, 2], [0.01, 0.002])
```

Matrices can also be built by converting arrays of numbers with uncertainties into matrices through the `unumpy.matrix` class:

```
>>> mat = unumpy.matrix(arr)
```

`unumpy.matrix` objects behave like `numpy.matrix` objects of numbers with uncertainties, but with better support for some operations (such as matrix inversion). For instance, regular NumPy matrices cannot be inverted, if they contain numbers with uncertainties (i.e., `numpy.matrix([[ufloat(...), ...]]).I` does not work). This is why the `unumpy.matrix` class is provided: both the inverse and the pseudo-inverse of a matrix can be calculated in the usual way: if `mat` is a `unumpy.matrix`,

```
>>> print mat.I
```

does calculate the inverse or pseudo-inverse of `mat` with uncertainties.

Uncertainties and nominal values

Nominal values and uncertainties in arrays (and matrices) can be directly accessed (through functions that work on pure float arrays too):

```
>>> unumpy.nominal_values(arr)
array([ 1.,  2.])
>>> unumpy.std_devs(mat)
matrix([[ 0.1  ,  0.002]])
```

Mathematical functions

This module defines uncertainty-aware mathematical functions that generalize those from `uncertainties.umath` so that they work on NumPy arrays of numbers with uncertainties instead of just scalars:

```
>>> print unumpy.cos(arr)  # Cosine of each array element
```

NumPy's function names are used, and not those from the `math` module (for instance, `unumpy.arccos()` is defined, like in NumPy, and is not named `acos()` like in the `math` module).

The definition of the mathematical quantities calculated by these functions is available in the documentation for `uncertainties.umath` (which is accessible through `help()` or `pydoc`).

NaN testing and NaN-aware operations

One particular function pertains to NaN testing: `unumpy.isnan()`. It returns true for each NaN *nominal value* (and false otherwise).

Since NaN±1 is *not* (the scalar) NaN, functions like `numpy.nanmean()` do not skip such values. This is where `unumpy.isnan()` is useful, as it can be used for masking out numbers with a NaN nominal value:

```
>>> nan = float("nan")
>>> arr = numpy.array([nan, uncertainties.ufloat(nan, 1), uncertainties.ufloat(1, nan),␣
↪2])
>>> arr
array([nan, nan+/-1.0, 1.0+/-nan, 2], dtype=object)
>>> arr[~unumpy.isnan(arr)].mean()
1.5+/-nan
```

or equivalently, by using masked arrays:

```
>>> masked_arr = numpy.ma.array(arr, mask=unumpy.isnan(arr))
>>> masked_arr.mean()
1.5+/-nan
```

In this case the uncertainty is NaN as it should be, because one of the numbers does have an undefined uncertainty, which makes the final uncertainty undefined (but the average is well defined). In general, uncertainties are not NaN and one obtains the mean of the non-NaN values.

1.3.3 Storing arrays in text format

Arrays of numbers with uncertainties can be directly *pickled*, saved to file and read from a file. Pickling has the advantage of preserving correlations between errors.

Storing instead arrays in **text format** loses correlations between errors but has the advantage of being both computer- and human-readable. This can be done through NumPy's `savetxt()` and `loadtxt()`.

Writing the array to file can be done by asking NumPy to use the *representation* of numbers with uncertainties (instead of the default float conversion):

```
>>> numpy.savetxt('arr.txt', arr, fmt='%r')
```

This produces a file *arr.txt* that contains a text representation of the array:

```
1.0+/-0.01
2.0+/-0.002
```

The file can then be read back by instructing NumPy to convert all the columns with *uncertainties.ufloat_fromstr()*. The number `num_cols` of columns in the input file (1, in our example) must be determined in advance, because NumPy requires a converter for each column separately. For Python 2:

```
>>> converters = dict.fromkeys(range(num_cols), uncertainties.ufloat_fromstr)
```

For Python 3, since `numpy.loadtxt()` passes bytes to converters, they must first be converted into a string:

```
>>> converters = dict.fromkeys(
        range(num_cols),
        lambda col_bytes: uncertainties.ufloat_fromstr(col_bytes.decode("latin1")))
```

(Latin 1 appears to in fact be the encoding used in `numpy.savetxt()` [as of NumPy 1.12]. This encoding seems to be the one hardcoded in `numpy.compat.asbytes()`.)

The array can then be loaded:

```
>>> arr = numpy.loadtxt('arr.txt', converters=converters, dtype=object)
```

1.3.4 Additional array functions: unumpy.ulinalg

The `unumpy.ulinalg` module contains more uncertainty-aware functions for arrays that contain numbers with uncertainties.

It currently offers generalizations of two functions from `numpy.linalg` that work on arrays (or matrices) that contain numbers with uncertainties, the **matrix inverse and pseudo-inverse**:

```
>>> unumpy.ulinalg.inv([[ufloat(2, 0.1)]])
array([[0.5+/-0.025]], dtype=object)
>>> unumpy.ulinalg.pinv(mat)
matrix([[0.2+/-0.0012419339757],
        [0.4+/-0.00161789987329]], dtype=object)
```

1.4 Formatting Variables with uncertainties

1.4.1 Printing

Numbers with uncertainties can be printed conveniently:

```
>>> print(x)
0.200+/-0.010
```

The resulting form can generally be parsed back with `ufloat_fromstr()` (except for the LaTeX form).

The nominal value and the uncertainty always have the **same precision**: this makes it easier to compare them.

Standard formats

More **control over the format** can be obtained (in Python 2.6+) through the usual `format()` method of strings:

```
>>> print('Result = {:10.2f}'.format(x))
Result =       0.20+/-      0.01
```

(Python 2.6 requires `'{0:10.2f}'` instead, with the usual explicit index. In Python 2.5 and earlier versions, `str.format()` is not available, but one can use the `format()` method of numbers with uncertainties instead: `'Result = %s' % x.format('10.2f')`.)

All the float format specifications are accepted, except those with the `n` format type. In particular, a fill character, an alignment option, a sign or zero option, a width, or the `%` format type are all supported.

The usual **float formats with a precision** retain their original meaning (e.g. `.2e` uses two digits after the decimal point): code that works with floats produces similar results when running with numbers with uncertainties.

Precision control

It is possible to **control the number of significant digits of the uncertainty** by adding the precision modifier `u` after the precision (and before any valid float format type like `f`, `e`, the empty format type, etc.):

```
>>> print('1 significant digit on the uncertainty: {:.1u}'.format(x))
1 significant digit on the uncertainty: 0.20+/-0.01
>>> print('3 significant digits on the uncertainty: {:.3u}'.format(x))
3 significant digits on the uncertainty: 0.2000+/-0.0100
>>> print('1 significant digit, exponent notation: {:.1ue}'.format(x))
1 significant digit, exponent notation: (2.0+/-0.1)e-01
>>> print('1 significant digit, percentage: {:.1u%}'.format(x))
1 significant digit, percentage: (20+/-1)%
```

When *uncertainties* must **choose the number of significant digits on the uncertainty**, it uses the Particle Data Group rounding rules (these rules keep the number of digits small, which is convenient for reading numbers with uncertainties, and at the same time prevent the uncertainty from being displayed with too few digits):

```
>>> print('Automatic number of digits on the uncertainty: {}'.format(x))
Automatic number of digits on the uncertainty: 0.200+/-0.010
>>> print(x0
0.200+/-0.010
```

Custom options

uncertainties provides even more flexibility through custom formatting options. They can be added at the end of the format string:

- `P` for **pretty-printing**:

  ```
  >>> print('{:.2e}'.format(x))
  (2.00+/-0.10)e-01
  >>> print(u'{:.2eP}'.format(x))
  (2.00±0.10)×10⁻¹
  ```

  The pretty-printing mode thus uses "±", "×" and superscript exponents. Note that the pretty-printing mode implies using **Unicode format strings** (`u'...'` in Python 2, but simply `'...'` in Python 3).

- `S` for the **shorthand notation**:

  ```
  >>> print('{:+.1uS}'.format(x))  # Sign, 1 digit for the uncertainty, shorthand
  +0.20(1)
  ```

  In this notation, the digits in parentheses represent the uncertainty on the last digits of the nominal value.

- `L` for a **LaTeX** output:

  ```
  >>> print x*1e7
  (2.00+/-0.10)e+06
  >>> print('{:L}'.format(x*1e7))  # Automatic exponent form, LaTeX
  \left(2.00 \pm 0.10\right) \times 10^{6}
  ```

- `p` is for requiring that parentheses be always printed around the ...±... part (without enclosing any exponent or trailing "%", etc.). This can for instance be useful so as to explicitly factor physical units:

```
>>> print('{:p} kg'.format(x))  # Adds parentheses
(0.200+/-0.010) kg
>>> print("{:p} kg".format(x*1e7))  # No parentheses added (exponent)
(2.00+/-0.10)e+06 kg
```

These custom formatting options **can be combined** (when meaningful).

Details

A **common exponent** is automatically calculated if an exponent is needed for the larger of the nominal value (in absolute value) and the uncertainty (the rule is the same as for floats). The exponent is generally **factored**, for increased legibility:

```
>>> print(x*1e7)
(2.00+/-0.10)e+06
```

When a *format width* is used, the common exponent is not factored:

```
>>> print('Result = {:10.1e}'.format(x*1e-10))
Result =    2.0e-11+/-   0.1e-11
```

(Using a (minimal) width of 1 is thus a way of forcing exponents to not be factored.) Thanks to this feature, each part (nominal value and standard deviation) is correctly aligned across multiple lines, while the relative magnitude of the error can still be readily estimated thanks to the common exponent.

An uncertainty which is *exactly* **zero** is always formatted as an integer:

```
>>> print(ufloat(3.1415, 0))
3.1415+/-0
>>> print(ufloat(3.1415e10, 0))
(3.1415+/-0)e+10
>>> print(ufloat(3.1415, 0.0005))
3.1415+/-0.0005
>>> print('{:.2f}'.format(ufloat(3.14, 0.001)))
3.14+/-0.00
>>> print('{:.2f}'.format(ufloat(3.14, 0.00)))
3.14+/-0
```

All the digits of a number with uncertainty are given in its representation:

```
>>> y = ufloat(1.23456789012345, 0.123456789)
>>> print(y)
1.23+/-0.12
>>> print(repr(y))
1.23456789012345+/-0.123456789
>>> y
1.23456789012345+/-0.123456789
```

More information on formatting can be obtained with `pydoc uncertainties.UFloat.__format__` (customization of the LaTeX output, etc.).

Global formatting

It is sometimes useful to have a **consistent formatting** across multiple parts of a program. Python's string.Formatter class allows one to do just that. Here is how it can be used to consistently use the shorthand notation for numbers with uncertainties:

```
class ShorthandFormatter(string.Formatter):

    def format_field(self, value, format_spec):
        if isinstance(value, uncertainties.UFloat):
            return value.format(format_spec+'S')  # Shorthand option added
        # Special formatting for other types can be added here (floats, etc.)
        else:
            # Usual formatting:
            return super(ShorthandFormatter, self).format_field(
                value, format_spec)

frmtr = ShorthandFormatter()

print(frmtr.format("Result = {0:.1u}", x))  # 1-digit uncertainty
```

prints with the shorthand notation: `Result = 0.20(1)`.

Customizing the pretty-print and LaTeX outputs

The pretty print and LaTeX outputs themselves can be customized.

For example, the pretty-print representation of numbers with uncertainty can display multiplication with a centered dot (·) instead of the default symbol (×), like in $(2.00 \pm 0.10) \cdot 10^{-1}$; this is easily done through the global setting `uncertainties.core.MULT_SYMBOLS["pretty-print"] = "·"`.

Beyond this multiplication symbol, the "±" symbol, the parentheses and the exponent representations can also be customized globally. The details can be found in the documentation of `uncertainties.core.format_num()`.

1.5 Technical Guide

1.5.1 Testing whether an object is a number with uncertainty

The recommended way of testing whether `value` carries an uncertainty handled by this module is by checking whether `value` is an instance of `UFloat`, through `isinstance(value, uncertainties.UFloat)`.

1.5.2 Pickling

The quantities with uncertainties created by the *uncertainties* package can be pickled (they can be stored in a file, for instance).

If multiple variables are pickled together (including when pickling *NumPy arrays*), their correlations are preserved:

```
>>> import pickle
>>> x = ufloat(2, 0.1)
>>> y = 2*x
```

(continues on next page)

(continued from previous page)

```
>>> p = pickle.dumps([x, y])  # Pickling to a string
>>> (x2, y2) = pickle.loads(p)  # Unpickling into new variables
>>> y2 - 2*x2
0.0+/-0
```

The final result is exactly zero because the unpickled variables `x2` and `y2` are completely correlated.

However, **unpickling necessarily creates new variables that bear no relationship with the original variables** (in fact, the pickled representation can be stored in a file and read from another program after the program that did the pickling is finished: the unpickled variables cannot be correlated to variables that can disappear). Thus

```
>>> x - x2
0.0+/-0.14142135623730953
```

which shows that the original variable `x` and the new variable `x2` are completely uncorrelated.

1.5.3 Comparison operators

Comparison operations (`>`, `==`, etc.) on numbers with uncertainties have a **pragmatic semantics**, in this package: numbers with uncertainties can be used wherever Python numbers are used, most of the time with a result identical to the one that would be obtained with their nominal value only. This allows code that runs with pure numbers to also work with numbers with uncertainties.

The **boolean value** (`bool(x)`, `if x ...`) of a number with uncertainty `x` is defined as the result of `x != 0`, as usual.

However, since the objects defined in this module represent probability distributions and not pure numbers, comparison operators are interpreted in a specific way.

The result of a comparison operation is defined so as to be essentially consistent with the requirement that uncertainties be small: the **value of a comparison operation** is True only if the operation yields True for all *infinitesimal* variations of its random variables around their nominal values, *except*, possibly, for an *infinitely small number* of cases.

Example:

```
>>> x = ufloat(3.14, 0.01)
>>> x == x
True
```

because a sample from the probability distribution of `x` is always equal to itself. However:

```
>>> y = ufloat(3.14, 0.01)
>>> x == y
False
```

since `x` and `y` are independent random variables that *almost* always give a different value (put differently, `x-y` is not equal to 0, as it can take many different values). Note that this is different from the result of `z = 3.14; t = 3.14; print(z == t)`, because `x` and `y` are *random variables*, not pure numbers.

Similarly,

```
>>> x = ufloat(3.14, 0.01)
>>> y = ufloat(3.00, 0.01)
>>> x > y
True
```

because `x` is supposed to have a probability distribution largely contained in the 3.14±~0.01 interval, while `y` is supposed to be well in the 3.00±~0.01 one: random samples of `x` and `y` will most of the time be such that the sample from `x` is larger than the sample from `y`. Therefore, it is natural to consider that for all practical purposes, `x > y`.

Since comparison operations are subject to the same constraints as other operations, as required by the *linear approximation* method, their result should be essentially *constant* over the regions of highest probability of their variables (this is the equivalent of the linearity of a real function, for boolean values). Thus, it is not meaningful to compare the following two independent variables, whose probability distributions overlap:

```
>>> x = ufloat(3, 0.01)
>>> y = ufloat(3.0001, 0.01)
```

In fact the function (x, y) → (x > y) is not even continuous over the region where x and y are concentrated, which violates the assumption of approximate linearity made in this package on operations involving numbers with uncertainties. Comparing such numbers therefore returns a boolean result whose meaning is undefined.

However, values with largely overlapping probability distributions can sometimes be compared unambiguously:

```
>>> x = ufloat(3, 1)
>>> x
3.0+/-1.0
>>> y = x + 0.0002
>>> y
3.0002+/-1.0
>>> y > x
True
```

In fact, correlations guarantee that `y` is always larger than `x`: `y-x` correctly satisfies the assumption of linearity, since it is a constant "random" function (with value 0.0002, even though `y` and `x` are random). Thus, it is indeed true that `y > x`.

1.5.4 Linear propagation of uncertainties

Constraints on the uncertainties

This package calculates the standard deviation of mathematical expressions through the linear approximation of error propagation theory.

The standard deviations and nominal values calculated by this package are thus meaningful approximations as long as **uncertainties are "small"**. A more precise version of this constraint is that the final calculated functions must have **precise linear expansions in the region where the probability distribution of their variables is the largest**. Mathematically, this means that the linear terms of the final calculated functions around the nominal values of their variables should be much larger than the remaining higher-order terms over the region of significant probability (because such higher-order contributions are neglected).

For example, calculating `x*10` with `x` = 5±3 gives a *perfect result* since the calculated function is linear. So does `umath.atan(umath.tan(x))` for `x` = 0±1, since only the *final* function counts (not an intermediate function like `tan()`).

Another example is `sin(0+/-0.01)`, for which *uncertainties* yields a meaningful standard deviation since the sine is quite linear over 0±0.01. However, `cos(0+/-0.01)`, yields an approximate standard deviation of 0 because it is parabolic around 0 instead of linear; this might not be precise enough for all applications.

More precise uncertainty estimates can be obtained, if necessary, with the soerp and mcerp packages. The soerp package performs *second-order* error propagation: this is still quite fast, but the standard deviation of higher-order functions like f(x) = x^3 for x = 0±0.1 is calculated as being exactly zero (as with *uncertainties*). The mcerp

package performs Monte-Carlo calculations, and can in principle yield very precise results, but calculations are much slower than with approximation schemes.

NaN uncertainty

If linear error propagation theory cannot be applied, the functions defined by *uncertainties* internally use a not-a-number value (`nan`) for the derivative.

As a consequence, it is possible for uncertainties to be `nan`:

```
>>> umath.sqrt(ufloat(0, 1))
0.0+/-nan
```

This indicates that **the derivative required by linear error propagation theory is not defined** (a Monte-Carlo calculation of the resulting random variable is more adapted to this specific case).

However, even in this case where the derivative at the nominal value is infinite, the *uncertainties* package **correctly handles perfectly precise numbers**:

```
>>> umath.sqrt(ufloat(0, 0))
0.0+/-0
```

is thus the correct result, despite the fact that the derivative of the square root is not defined in zero.

1.5.5 Mathematical definition of numbers with uncertainties

Mathematically, **numbers with uncertainties** are, in this package, **probability distributions**. They are *not restricted* to normal (Gaussian) distributions and can be **any distribution**. These probability distributions are reduced to two numbers: a nominal value and an uncertainty.

Thus, both independent variables (`Variable` objects) and the result of mathematical operations (`AffineScalarFunc` objects) contain these two values (respectively in their `nominal_value` and `std_dev` attributes).

The **uncertainty** of a number with uncertainty is simply defined in this package as the **standard deviation** of the underlying probability distribution.

The numbers with uncertainties manipulated by this package are assumed to have a probability distribution mostly contained around their nominal value, in an interval of about the size of their standard deviation. This should cover most practical cases.

A good choice of **nominal value** for a number with uncertainty is thus the median of its probability distribution, the location of highest probability, or the average value.

Probability distributions (random variables and calculation results) are printed as:

```
nominal value +/- standard deviation
```

but this does not imply any property on the nominal value (beyond the fact that the nominal value is normally inside the region of high probability density), or that the probability distribution of the result is symmetrical (this is rarely strictly the case).

1.5.6 Differentiation method

The *uncertainties* package automatically calculates the derivatives required by linear error propagation theory.

Almost all the derivatives of the fundamental functions provided by *uncertainties* are obtained through analytical formulas (the few mathematical functions that are instead differentiated through numerical approximation are listed in `umath_core.num_deriv_funcs`).

The derivatives of mathematical *expressions* are evaluated through a fast and precise method: *uncertainties* transparently implements automatic differentiation with reverse accumulation. This method essentially consists in keeping track of the value of derivatives, and in automatically applying the chain rule. Automatic differentiation is faster than symbolic differentiation and more precise than numerical differentiation.

The derivatives of any expression can be obtained with *uncertainties* in a simple way, as demonstrated in the *User Guide*.

1.5.7 Tracking of random variables

This package keeps track of all the random variables a quantity depends on, which allows one to perform transparent calculations that yield correct uncertainties. For example:

```
>>> x = ufloat(2, 0.1)
>>> a = 42
>>> poly = x**2 + a
>>> poly
46.0+/-0.4
>>> poly - x*x
42+/-0
```

Even though `x*x` has a non-zero uncertainty, the result has a zero uncertainty, because it is equal to `a`.

If the variable `a` above is modified, the value of `poly` is not modified, as is usual in Python:

```
>>> a = 123
>>> print poly
46.0+/-0.4  # Still equal to x**2 + 42, not x**2 + 123
```

Random variables can, on the other hand, have their uncertainty updated on the fly, because quantities with uncertainties (like `poly`) keep track of them:

```
>>> x.std_dev = 0
>>> print poly
46+/-0  # Zero uncertainty, now
```

As usual, Python keeps track of objects as long as they are used. Thus, redefining the value of `x` does not change the fact that `poly` depends on the quantity with uncertainty previously stored in `x`:

```
>>> x = 10000
>>> print poly
46+/-0  # Unchanged
```

These mechanisms make quantities with uncertainties behave mostly like regular numbers, while providing a fully transparent way of handling correlations between quantities.

1.5.8 Python classes for variables and functions with uncertainty

Numbers with uncertainties are represented through two different classes:

1. a class for independent random variables (`Variable`, which inherits from `UFloat`),
2. a class for functions that depend on independent variables (`AffineScalarFunc`, aliased as `UFloat`).

Documentation for these classes is available in their Python docstring, which can for instance displayed through pydoc.

The factory function `ufloat()` creates variables and thus returns a `Variable` object:

```
>>> x = ufloat(1, 0.1)
>>> type(x)
<class 'uncertainties.Variable'>
```

`Variable` objects can be used as if they were regular Python numbers (the summation, etc. of these objects is defined).

Mathematical expressions involving numbers with uncertainties generally return `AffineScalarFunc` objects, because they represent mathematical functions and not simple variables; these objects store all the variables they depend on:

```
>>> type(umath.sin(x))
<class 'uncertainties.AffineScalarFunc'>
```

PYTHON MODULE INDEX

INDEX

T

U

V

W