uncertainties Python package Documentation

Release 3.0.3

Eric O. LEBIGOT (EOL)

Oct 28, 2018

Contents

CHAPTER 1

Welcome to the uncertainties package

The uncertainties package is a free, cross-platform program that **transparently** handles calculations with **numbers with uncertainties** (like 3.14±0.01). It can also yield the **derivatives** of any expression.

The `uncertainties` package **takes the pain and complexity out** of uncertainty calculations. Error propagation is not to be feared anymore!

Calculations of results with uncertainties, or of derivatives, can be performed either in an **interactive session** (as with a calculator), or in **programs** written in the Python programming language. Existing calculation code can **run with little or no change**.

Whatever the complexity of a calculation, this package returns its result with an uncertainty as predicted by linear error propagation theory. It automatically *calculates derivatives* and uses them for calculating uncertainties. Almost all uncertainty calculations are performed **analytically**.

Correlations between variables are automatically handled, which sets this module apart from many existing error propagation codes.

You may want to check the following related uncertainty calculation Python packages to see if they better suit your needs: soerp (higher-order approximations) and mcerp (Monte-Carlo approach).

1.1 An easy-to-use calculator

Calculations involving **numbers with uncertainties** can be performed even without knowing anything about the Python programming language. After *installing this package* and invoking the Python interpreter, calculations with **automatic error propagation** can be performed **transparently** (i.e., through the usual syntax for mathematical formulas):

```
>>> from uncertainties import ufloat
>>> from uncertainties.umath import *  # sin(), etc.
>>> x = ufloat(1, 0.1)  # x = 1+/-0.1
>>> print 2*x
2.00+/-0.20
```

(continues on next page)

(continued from previous page)

```
>>> sin(2*x)  # In a Python shell, "print" is optional
0.9092974268256817+/-0.08322936730942848
```

Thus, existing calculation code designed for regular numbers can run with numbers with uncertainties with *no or little modification*.

Another strength of this package is its correct handling of **correlations**. For instance, the following quantity is exactly zero even though `x` has an uncertainty:

```
>>> x-x
0.0+/-0
```

Many other error propagation codes return the incorrect value 0±0.1414… because they wrongly assume that the two subtracted quantities are *independent* random variables.

Arrays of numbers with uncertainties are *transparently handled* too.

Derivatives are similarly very *easy to obtain*:

```
>>> (2*x+1000).derivatives[x]
2.0
```

They are calculated with a *fast method*.

1.2 Available documentation

The *User Guide* details many of the features of this package.

The part *Uncertainties in arrays* describes how arrays of numbers with uncertainties can be created and used.

The *Technical Guide* gives advanced technical details.

Additional information is available through the pydoc command, which gives access to many of the documentation strings included in the code.

1.3 Installation and download

1.3.1 Important note

The installation commands below should be **run in a DOS or Unix command shell** (*not* in a Python shell).

Under Windows (version 7 and earlier), a command shell can be obtained by running `cmd.exe` (through the Run… menu item from the Start menu). Under Unix (Linux, Mac OS X,…), a Unix shell is available when opening a terminal (in Mac OS X, the Terminal program is found in the Utilities folder, which can be accessed through the Go menu in the Finder).

1.3.2 Automatic install or upgrade

One of the automatic installation or upgrade procedures below might work on your system, if you have a Python package installer or use certain Linux distributions.

Under Unix, it may be necessary to prefix the commands below with `sudo`, so that the installation program has **sufficient access rights to the system**.

If you use the Anaconda distribution, you can install the latest version with

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

If you have **'pip <hhttps://pypi.org/project/pip/'_**, you can try to install the latest version with

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

If you have setuptools, you can try to automatically install or upgrade this package with

```
easy_install --upgrade uncertainties
```

The `uncertainties` package is also available for **Windows** through the Python(x,y) distribution. It may also be included in Christoph Gohlke's Base distribution of scientific Python packages.

Mac OS X users who use the MacPorts package manager can install `uncertainties` with `sudo port install py**-uncertainties`, and upgrade it with `sudo port upgrade py**-uncertainties` where `**` represents the desired Python version (`27`, `33`, etc.).

The `uncertainties` package is also available through the following **Linux** distributions and software platforms: Ubuntu, Fedora, openSUSE, Debian and Maemo.

1.3.3 Manual download and install

Alternatively, you can simply download the package archive from the Python Package Index (PyPI) and unpack it. The package can then be installed by **going into the unpacked directory** (`uncertainties-...`), and running the provided `setup.py` program with

```
python setup.py install
```

(where the default `python` interpreter must generally be replaced by the version of Python for which the package should be installed: `python3`, `python3.3`, etc.).

For an installation with Python 2.6+ in the *user* Python library (no additional access rights needed):

```
python setup.py install --user
```

For an installation in a custom directory `my_directory`:

```
python setup.py install --install-lib my_directory
```

If additional access rights are needed (Unix):

```
sudo python setup.py install
```

You can also simply **move** the `uncertainties-py*` directory that corresponds best to your version of Python to a location that Python can import from (directory in which scripts using `uncertainties` are run, etc.); the chosen `uncertainties-py*` directory should then be renamed `uncertainties`. Python 3 users should then run `2to3 -w .` from inside this directory so as to automatically adapt the code to Python 3.

1.3.4 Source code

The latest, bleeding-edge but working code and documentation source are available on GitHub. The `uncertainties` package is written in pure Python and has no external dependency (the NumPy package is optional). It contains about 7000 lines of code. 75 % of these lines are documentation strings and comments. The remaining 25 % are split between unit tests (15 % of the total) and the calculation code proper (10 % of the total). `uncertainties` is thus a **lightweight, portable package** with abundant documentation and tests.

1.4 Migration from version 1 to version 2

Some **incompatible changes** were introduced in version 2 of `uncertainties` (see the version history). While the version 2 line will support the version 1 syntax for some time, it is recommended to **update existing programs** as soon as possible. This can be made easier through the provided **automatic updater**.

The automatic updater works like Python's 2to3 updater. It can be run (in a Unix or DOS shell) with:

```
python -m uncertainties.1to2
```

For example, updating a single Python program can be done with

```
python -m uncertainties.1to2 -w example.py
```

All the Python programs contained under a directory `Programs` (including in nested sub-directories) can be automatically updated with

```
python -m uncertainties.1to2 -w Programs
```

Backups are automatically created, unless the `-n` option is given.

Some **manual adjustments** might be necessary after running the updater (incorrectly modified lines, untouched obsolete syntax).

While the updater creates backup copies by default, it is generally useful to **first create a backup** of the modified directory, or alternatively to use some version control system. Reviewing the modifications with a file comparison tool might also be useful.

1.5 What others say

- "*Superb,*" "*wonderful,*" "*It's like magic.*" (Joaquin Abian)
- "*pretty amazing*" (John Kitchin)
- "*An awesome python package*" (Jason Moore)
- "*Utterly brilliant*" (Jeffrey Simpson)
- "*An amazing time saver*" (Paul Nakroshis)
- "*Seems to be the gold standard for this kind of thing*" (Peter Williams)
- "*This package has a great interface and makes error propagation something to stop fearing.*" (Dr Dawes)
- "*uncertainties makes error propagation dead simple.*" (enrico documentation)
- "*many inspiring ideas*" (Abraham Lee)
- "*Those of us working with experimental data or simulation results will appreciate this.*" (Konrad Hinsen)
- "*PyPI's uncertainties rocks!*" (Siegfried Gevatter)
- "*A very cool Python module*" (Ram Rachum)
- "*Holy f*** this would have saved me so much f***ing time last semester.*" (reddit)

1.6 Future developments

Planned future developments include (starting from the most requested ones):

- handling of complex numbers with uncertainties;
- increased support for NumPy: Fourier Transform with uncertainties, automatic wrapping of functions that accept or produce arrays, standard deviation of arrays, more convenient matrix creation, new linear algebra methods (eigenvalue and QR decompositions, determinant,. . .), input of arrays with uncertainties as strings (like in NumPy),. . . ;
- JSON support;
- addition of `real` and `imag` attributes, for increased compatibility with existing code (Python numbers have these attributes);
- addition of new functions from the `math` module;
- fitting routines that conveniently handle data with uncertainties;
- a re-correlate function that puts correlations back between data that was saved in separate files;
- support for multi-precision numbers with uncertainties.

Call for contributions: I got multiple requests for complex numbers with uncertainties, Fourier Transform support, and the automatic wrapping of functions that accept or produce arrays. Please contact me if you are interested in contributing. Patches are welcome. They must have a high standard of legibility and quality in order to be accepted (otherwise it is always possible to create a new Python package by branching off this one, and I would still be happy to help with the effort).

Please support the continued development of this program by donating $10 or more through PayPal (no PayPal account necessary). I love modern board games, so this will go towards giving my friends and I some special gaming time!

1.7 Contact

Feature requests, bug reports, or feedback are much welcome. They can be sent to the creator of `uncertainties`, Eric O. LEBIGOT (EOL).

1.8 How to cite this package

If you use this package for a publication (in a journal, on the web, etc.), please cite it by including as much information as possible from the following: *Uncertainties: a Python package for calculations with uncertainties*, Eric O. LEBIGOT. Adding the version number is optional.

1.9 Acknowledgments

The author wishes to thank all the people who made generous donations: they help keep this project alive by providing positive feedback.

I greatly appreciate 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 and Roman Yurchak are gratefully acknowledged.

I would also like to thank 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.

I am 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.

1.10 License

This software is released under a **dual license**; one of the following options can be chosen:

1. The Revised BSD License (© 2010–2018, Eric O. LEBIGOT [EOL]).
2. Any other license, as long as it is obtained from the creator of this package.

CHAPTER 2

User Guide

2.1 Basic setup

Basic mathematical operations involving numbers with uncertainties only require a simple import:

```
>>> from uncertainties import ufloat
```

The `ufloat()` function creates numbers with uncertainties. Existing calculation code can usually run with no or little modification and automatically produce results with uncertainties.

The `uncertainties` module contains other features, which can be made accessible through

```
>>> import uncertainties
```

The `uncertainties` package also contains sub-modules for *advanced mathematical functions*, and *arrays and matrices*.

2.2 Creating numbers with uncertainties

Numbers with uncertainties can be input either numerically, or through one of many string representations, so that files containing numbers with uncertainties can easily be parsed. Thus, x = 0.20±0.01 can be expressed in many convenient ways, including:

```
>>> x = ufloat(0.20, 0.01)  # x = 0.20+/-0.01
```

```
>>> from uncertainties import ufloat_fromstr
>>> 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
```

Each number created this way is an **independent (random) variable** (for details, see the *Technical Guide*).

More information can be obtained with `pydoc uncertainties.ufloat` and `pydoc uncertainties.ufloat_fromstr` ("20(1)×10^{-2}" is also recognized, etc.).

2.3 Basic math

Calculations can be performed directly, as with regular real numbers:

```
>>> square = x**2
>>> print square
0.040+/-0.004
```

2.4 Mathematical operations

Besides being able to apply basic mathematical operations to numbers with uncertainty, this package provides generalizations of **most of the functions from the standard `math` module**. These mathematical functions are found in the `uncertainties.umath` module:

```
>>> from uncertainties.umath import *  # Imports sin(), etc.
>>> sin(x**2)
0.03998933418663417+/-0.003996800426643912
```

The list of available mathematical functions can be obtained with the `pydoc uncertainties.umath` command.

2.4.1 NaN testing

NaN values can appear in a number with uncertainty. Care must be taken with such values, as values like NaN±1, 1±NaN and NaN±NaN are by definition *not* NaN, which is a float.

Testing whether a number with uncertainty has a **NaN nominal value** can be done with the provided function `uncertainties.umath.isnan()`, which generalizes the standard `math.isnan()`.

Checking whether the *uncertainty* of `x` is NaN can be done directly with the standard function: `math.isnan(x.std_dev)` (or equivalently `math.isnan(x.s)`).

2.5 Arrays of numbers with uncertainties

It is possible to put numbers with uncertainties 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
```

Thus, usual operations on NumPy arrays can be performed transparently even when these arrays contain numbers with uncertainties.

More complex operations on NumPy arrays and matrices can be performed through the dedicated `uncertainties.unumpy` module.

2.6 Correlated variables

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 completely **transparently**.

2.7 Access to the uncertainty and to the nominal value

The nominal value and the uncertainty (standard deviation) can also be accessed independently:

```
>>> print square
0.040+/-0.004
>>> print square.nominal_value
0.04
>>> print square.n   # Abbreviation
0.04
>>> print square.std_dev
0.004
>>> print square.s   # Abbreviation
0.004
```

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)
```

(continues on next page)

(continued from previous page)

```
...
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.

2.9 Comparison operators

Comparison operators behave in a natural way:

```
>>> print x
0.200+/-0.010
>>> y = x + 0.0001
>>> y
0.2001+/-0.01
>>> y > x
True
>>> y > 0
True
```

One important concept to keep in mind is that `ufloat()` creates a random variable, so that two numbers with the same nominal value and standard deviation are generally different:

```
>>> y = ufloat(1, 0.1)
>>> z = ufloat(1, 0.1)
>>> print y
1.00+/-0.10
>>> print z
1.00+/-0.10
>>> y == y
True
>>> y == z
False
```

In physical terms, two rods of the same nominal length and uncertainty on their length are generally of different sizes: `y` is different from `z`.

More detailed information on the semantics of comparison operators for numbers with uncertainties can be found in the *Technical Guide*.

2.10 Covariance and correlation matrices

2.10.1 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
```

2.10.2 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.        ]])
```

2.11 Correlated variables

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

2.11.1 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).

2.11.2 Use of a correlation matrix

Alternatively, correlated values can be defined through a *correlation* matrix (the correlation matrix is the covariance matrix normalized with individual standard deviations; it has ones on its diagonal), along with a list of nominal values and standard deviations:

```
>>> (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()`).

2.12 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.

2.12.1 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.

2.12.2 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 x
0.200+/-0.010
```

2.12.3 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}
  ```

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

2.12.4 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.).

2.12.5 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.)
```

(continues on next page)

(continued from previous page)

```
        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)`.

2.13 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()` *accepts numbers with uncertainties* as arguments *wherever a Python float is used* for `f()`. `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).

2.14 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
```

2.15 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*.

2.16 Additional information

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

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

CHAPTER 3

Uncertainties in arrays

3.1 The unumpy package

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.

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

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

These features can be made available with

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

3.1.1 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]])
```

3.1.2 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.

3.2 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)
```

3.3 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)
```

CHAPTER 4

Technical Guide

4.1 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)`.

4.2 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
>>> 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.

4.3 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
True
```

since `x` and `y` are independent random variables that *almost* always give a different value. 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 $(x, y) \to (x > y)$ 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`.

4.4 Linear propagation of uncertainties

4.4.1 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 $x = 0\pm0.1$ 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.

4.4.2 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.

4.5 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).

4.6 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*.

4.7 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.

4.8 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'>
```

Index