GUM Tree Calculator Documentation

Release 1.1.0

Measurement Standards Laboratory of New Zealand

May 30, 2019

Contents

CHAPTER 1

Introduction

1.1 Installing GTC

1.1.1 From PyPI

GTC is available as a PyPI package. It can be installed using pip

```
pip install gtc
```

This obtains the most recent stable release of **GTC** and is the recommended way to install the package.

1.1.2 From the Source Code

GTC is actively developed on GitHub, where the source code is available.

The easiest way to install **GTC** with the latest features and updates is to run

```
pip install https://github.com/MSLNZ/GTC/archive/master.zip
```

Alternatively, you can either clone the public repository

```
git clone git://github.com/MSLNZ/GTC.git
```

or download the tarball (Unix) or zipball (Windows) and then extract it.

Once you have a copy of the source code, you can install it by running

```
cd GTC
pip install .
```

1.1.3 Dependencies

- Python 2.7, 3.5+
- scipy

1.2 Introduction

The GUM Tree calculator (`GTC`) is a data processing tool that uses *uncertain numbers* to represent measured quantities. `GTC` automates evaluation of uncertainty in derived quantities when they are calculated from measured data.

1.2.1 Measurement error

A measurement obtains information about a quantity, but the quantity itself (the *measurand*) is never determined exactly. There is always some *measurement error* involved. This can be expressed as an equation, where the unknown measurand is Y and the measurement result is y, we have

$$y = Y + E_y \ ,$$

where E_y is the measurement error. So, the result, y, is only an approximate value for the quantity of interest Y.

This is how 'uncertainty' arises. After any measurement, we are faced with uncertainty about what will happen if we take the measured value y and use it for the (unknown) value Y.

For example, suppose the speed of a car is measured by a law enforcement officer. The officer needs to decide whether, in fact, a car was travelling faster than the legal limit but this simple fact cannot be determined, because the actual speed Y remains unknown. The measured value y might indicate that the car was speeding when in fact it was not, or that it was not speeding when in fact it was. In practice, a decision rule that takes account of the measurement uncertainty must be used. In this example, the rule will probably err on the side of caution (a few speeding drivers will escape rather than unfairly accusing good drivers of speeding).

Like the measurand, the measurement error E_y will never be known. At best, its behaviour can be described in statistical terms. This leads to technical meanings of the word 'uncertainty'. For instance, the term 'standard uncertainty' refers to the standard deviation of a statistical distribution associated with an unpredictable quantity.

Measurement models

A measurement error comes about because there are unpredictable factors that influence the outcome of a measurement process. In a formal analysis, these factors must be identified and included in a measurement model, which defines the measurand in terms of all other significant influence quantities. In mathematical terms, we write

$$Y = f(X_1, X_2, \cdots) \ ,$$

where the X_i are influence quantities.

Once again, the actual quantities $X_1, X_2, \cdots$ are not known; only estimates $x_1, x_2, \cdots$ are available. These are used to calculate a measured value that is approximately equal to the measurand

$$y = f(x_1, x_2, \cdots) \; .$$

1.2.2 Uncertain Numbers

An uncertain number is a data-type designed to represent a measured quantity. It encapsulates information about the measurement, including the measured value and its uncertainty.

Uncertain numbers are used when processing measurement data; that is, to evaluate measurement models. The inputs to a model (like $X_1, X_2, \cdots$ above) will be defined as uncertain numbers using measurement data. Calculations then produce an uncertain number for the measurand (Y).

There are two types of uncertain number: one for real-valued quantities and one for complex-valued quantities. At the very least, two pieces of information are needed to define an uncertain number: a value (that is, a measured, or approximate, value of the quantity) and the uncertainty associated with the error in the measured value.

Uncertain real numbers

The function `ureal()` is usually the preferred way to define uncertain numbers representing real-valued quantities.

Example: an electrical circuit

Suppose the current flowing in an electrical circuit I and the voltage across a circuit element V have been measured.

The measured values are $x_V = 0.1\,\mathrm{V}$ and $x_I = 15\,\mathrm{mA}$, with standard uncertainties $u(x_V) = 1\,\mathrm{mV}$ and $u(x_I) = 0.5\,\mathrm{mA}$, respectively.

Uncertain numbers for V and I are defined by

```
>>> V = ureal(0.1,1E-3)
>>> I = ureal(15E-3,0.5E-3)
```

and then the resistance can be calculated directly using Ohm's law

```
>>> R = V/I
>>> print(R)
6.67(23)
```

The measured value of resistance $x_R = 6.67\,\Omega$ is an estimate (approximation) for R, the standard uncertainty in x_R as an estimate of R is $0.23\,\Omega$.

Example: height of a flag pole

Suppose a flag is flying from a pole that is 15 metres away from an observer (with an uncertainty of 3 cm). The angle between horizontal and line-of-sight to the top of the pole is 38 degrees (with an uncertainty of 2 degrees). How high is the top of the pole?

A measurement model should express a relationship between the quantities involved: the height of the pole H, the distance to the base of the pole B and the line-of-sight angle Φ,

$$H = B \tan \Phi \; .$$

To calculate the height, we create uncertain numbers representing the measured quantities and use the model

```
>>> B = ureal(15,3E-2)
>>> Phi = ureal(math.radians(38),math.radians(2))
>>> H = B * tan(Phi)
>>> print(H)
11.72(84)
```

The result $x_H = 11.7$ metres is our best estimate of the height H. The standard uncertainty of this value, as an estimate of the actual height, is 0.8 metres.

It is important to note that uncertain-number calculations are open ended. In this case, for example, we can keep going and evaluate what the observer angle would be at 20 metres from the pole (the uncertainty in the base distance remains 3 cm)

```
>>> B_20 = ureal(20,3E-2)
>>> Phi_20 = atan( H/B_20 )
>>> print(Phi_20)
0.530(31)
>>> Phi_20_deg= Phi_20 * 180./math.pi
>>> print(Phi_20_deg)
30.4(1.8)
```

The angle of 30.4 degrees at 20 metres from the pole has a standard uncertainty of 1.8 degrees.

Uncertain complex numbers

The function `ucomplex()` is usually preferred for defining uncertain complex numbers.

Example: AC electric circuit

Suppose measurements have been made of: the alternating current i flowing in an electrical circuit, the voltage v across a circuit element and the phase ϕ of the voltage with respect to the current. The measured values are: $x_v \approx 4.999\,\mathrm{V}$, $x_i \approx 19.661\,\mathrm{mA}$ and $x_\phi \approx 1.04446\,\mathrm{rad}$, with standard uncertainties $u(x_v) = 0.0032\,\mathrm{V}$, $u(x_i) = 0.0095\,\mathrm{mA}$ and $u(x_\phi) = 0.00075\,\mathrm{rad}$, respectively.

Uncertain numbers for the quantities v, i and ϕ can be defined

```
>>> v = ucomplex(complex(4.999,0),(0.0032,0))
>>> i = ucomplex(complex(19.661E-3,0),(0.0095E-3,0))
>>> phi = ucomplex(complex(0,1.04446),(0,0.00075))
```

Note, the uncertainty argument is a pair of numbers in these definitions. These are the standard uncertainties associated with measured values of the real and imaginary components.

The complex impedance is

```
>>> z = v * exp(phi) / i
>>> print(z)
(127.73(19)+219.85(20)j)
```

We see that our best estimate of the impedance is the complex value $(127.73 + \mathrm{j}219.85)\,\Omega$. The standard uncertainty in the real component is $0.19\,\Omega$ and the standard uncertainty in the imaginary component is $0.20\,\Omega$. There is also a small correlation between our estimates of the real and imaginary components

```
>>> get_correlation(z)
0.0582038103158399...
```

If a polar representation of the impedance is preferred,

```
>>> print(magnitude(z))
254.26(20)
>>> print(phase(z))
1.04446(75)
```

Uncertain Number Attributes

Uncertain number objects have attributes that provide access to: the measured value (the estimate), the uncertainty (of the estimate) and the degrees of freedom (associated with the uncertainty) (see *UncertainReal*).

Continuing with the flagpole example, the attributes `x`, `u`, `df` obtain the value, the uncertainty and the degrees-of-freedom (which is infinity), respectively

```
>>> H.x
11.71928439760076...
>>> H.u
0.84353295110757...
>>> H.df
inf
```

Alternatively, there are functions that return the same attributes

```
>>> value(H)
11.71928439760076...
>>> uncertainty(H)
0.84353295110757...
>>> dof(H)
inf
```

Uncertain numbers and measurement errors

It is often is helpful to to formulate measurement models that explicitly acknowledge measurement errors. As we said above, these errors are not known exactly; many will be residual quantities with estimates of zero or unity. However, errors have a physical meaning and it is often useful to identify them in the model.

In the example above, errors associated with measured values of B and Φ were not identified but we can do so now by introducing the terms E_b and E_ϕ. The measured values $b = 15\,\mathrm{m}$ and $\phi = 38\,\mathrm{deg}$ are related to the quantities of interest as

$$B = b - E_b$$
$$\Phi = \phi - E_\phi$$

Our best estimates of these errors are trivial, $E_b \approx 0$ and $E_\phi \approx 0$, but the actual values are unpredictable and give rise to uncertainty in the height of the pole. It is appropriate to attribute the standard uncertainties $u(E_b) = 3 \times 10^2\,\mathrm{m}$ and $u(E_\phi) = 2\,\mathrm{deg}$ to measurement errors, rather than associate uncertainty with the fixed quantities B and Φ.

The calculation becomes

```
>>> B = 15 - ureal(0,3E-2,label='E_b')
>>> Phi = math.radians(38) - ureal(0,math.radians(2),label='E_phi')
>>> H = B*tan(Phi)
>>> print(H)
11.72(84)
```

This reflects our understanding of the problem better: the numbers $b = 15$ and $\phi = 38$ are known, there is nothing 'uncertain' about their values. What is uncertain are the unknown measurement errors E_b and E_ϕ.

When defining uncertain numbers, setting labels allows an uncertainty budget to be displayed later (see *budget()*). For instance,

```
>>> for cpt in rp.budget(H):
...     print("{0.label}: {0.u:.3f}".format(cpt))
...
E_phi: 0.843
E_b: 0.023
```

CHAPTER 2

GTC Modules

2.1 Core Functions and Classes

2.1.1 Core Functions

Functions that create elementary uncertain numbers and functions that access uncertain-number attributes, are defined in the `core` module. There is also a set of standard mathematical functions (e.g.: `sqrt()`, `sin()`, `log10()`, etc) for uncertain numbers. These functions can be applied to the numeric Python types too.

All `core` functions are automatically imported into the `GTC` namespace (i.e., they are available after `from GTC import *`).

ureal (*x*, *u*, *df=inf*, *label=None*, *independent=True*)

Create an elementary uncertain real number

Parameters

- **x** (`float`) – the value (estimate)
- **u** (`float`) – the standard uncertainty
- **df** (`float`) – the degrees-of-freedom
- **label** (`str`) – a string label
- **independent** (`bool`) – not correlated with other UNs

Return type `UncertainReal`

Example:

```
>>> ur = ureal(2.5,0.5,3,label='x')
>>> ur
ureal(2.5,0.5,3.0, label='x')
```

multiple_ureal (*x_seq*, *u_seq*, *df*, *label_seq=None*)

Return a sequence of related elementary uncertain real numbers

Parameters

- **x_seq** – a sequence of values (estimates)
- **u_seq** – a sequence of standard uncertainties
- **df** – the degrees-of-freedom
- **label_seq** – a sequence of labels

Return type a sequence of *UncertainReal*

Defines an set of uncertain real numbers with the same number of degrees-of-freedom.

Correlation between any pairs of this set of uncertain numbers defined will not invalidate degrees-of-freedom calculations. (see: R Willink, *Metrologia* 44 (2007) 340-349, Sec. 4.1)

Example:

```
# Example from GUM-H2
>>> x = [4.999,19.661E-3,1.04446]
>>> u = [3.2E-3,9.5E-6,7.5E-4]
>>> labels = ['V','I','phi']
>>> v,i,phi = multiple_ureal(x,u,4,labels)

>>> set_correlation(-0.36,v,i)
>>> set_correlation(0.86,v,phi)
>>> set_correlation(-0.65,i,phi)

>>> r = v/i*cos(phi)
>>> r
ureal(127.732169928102...,0.0699787279883717...,4.0)
```

multiple_ucomplex (*x_seq*, *u_seq*, *df*, *label_seq=None*)

Return a sequence of uncertain complex numbers

Parameters

- **x_seq** – a sequence of complex values
- **u_seq** – a sequence of standard uncertainties or covariances
- **df** – the degrees-of-freedom
- **label_seq** – a sequence of labels for the uncertain numbers

Return type a sequence of *UncertainComplex*

This function defines an set of uncertain complex numbers with the same number of degrees-of-freedom.

Correlation between any pairs of these uncertain numbers will not invalidate degrees-of-freedom calculations. (see: R Willink, *Metrologia* 44 (2007) 340-349, Sec. 4.1)

Example:

```
# GUM Appendix H2
>>> values = [4.999+0j,0.019661+0j,1.04446j]
>>> uncert = [(0.0032,0.0),(0.0000095,0.0),(0.0,0.00075)]
>>> v,i,phi = multiple_ucomplex(values,uncert,5)

>>> set_correlation(-0.36,v.real,i.real)
```

(continues on next page)

(continued from previous page)

```
>>> set_correlation(0.86,v.real,phi.imag)
>>> set_correlation(-0.65,i.real,phi.imag)

>>> z = v * exp(phi)/ i
>>> print(z)
(127.732(70)+219.847(296)j)
>>> z.r
-28.5825760885182...
```

ucomplex (*z*, *u*, *df=inf*, *label=None*, *independent=True*)

Create an elementary uncertain complex number

Parameters

- **z** (*complex*) – the value (estimate)
- **u** (*float, 2-element or 4-element sequence*) – the standard uncertainty or variance
- **df** (*float*) – the degrees-of-freedom

Return type *UncertainComplex*

Raises `ValueError` if `df` or `u` have illegal values.

`u` can be a float, a 2-element or 4-element sequence.

If `u` is a float, the standard uncertainty in both the real and imaginary components is taken to be `u`.

If `u` is a 2-element sequence, the first element is taken to be the standard uncertainty in the real component and the second element is taken to be the standard uncertainty in the imaginary component.

If `u` is a 4-element sequence, the sequence is interpreted as a variance-covariance matrix.

Examples:

```
>>> uc = ucomplex(1+2j,(.5,.5),3,label='x')
>>> uc
ucomplex((1+2j), u=[0.5,0.5], r=0.0, df=3.0, label=x)
```

```
>>> cv = (1.2,0.7,0.7,2.2)
>>> uc = ucomplex(0.2-.5j, cv)
>>> variance(uc)
VarianceCovariance(rr=1.1999999999999997, ri=0.7, ir=0.7, ii=2.2)
```

constant (*x*, *label=None*)

Create a constant uncertain number (with no uncertainty)

Parameters **x** (*float or complex*) – a number

Return type *UncertainReal* or *UncertainComplex*

If `x` is complex, return an uncertain complex number.

If `x` is real return an uncertain real number.

Example:

```
>>> e = constant(math.e,label='Euler')
>>> e
ureal(2.718281828459045,0.0,inf, label='Euler')
```

value (*x*)

Return the value

Returns a complex number if `x` is an uncertain complex number

Returns a real number if x is an uncertain real number

Returns x otherwise.

Example:

```
>>> un = ureal(3,1)
>>> value(un)
3.0
>>> un.x
3.0
```

uncertainty (*x*)

Return the standard uncertainty

If x is an uncertain complex number, return a 2-element sequence containing the standard uncertainties of the real and imaginary components.

If x is an uncertain real number, return the standard uncertainty.

Otherwise, return 0.

Examples:

```
>>> ur = ureal(2.5,0.5,3,label='x')
>>> uncertainty(ur)
0.5
>>> ur.u
0.5

>>> uc = ucomplex(1+2j,(.5,.5),3,label='x')
>>> uncertainty(uc)
StandardUncertainty(real=0.5, imag=0.5)
```

variance (*x*)

Return the standard variance

If x is an uncertain real number, return the standard variance.

If x is an uncertain complex number, return a 4-element sequence containing elements of the variance-covariance matrix.

Otherwise, return 0.

Examples:

```
>>> ur = ureal(2.5,0.5,3,label='x')
>>> variance(ur)
0.25
>>> ur.v
0.25

>>> uc = ucomplex(1+2j,(.5,.5),3,label='x')
>>> variance(uc)
VarianceCovariance(rr=0.25, ri=0.0, ir=0.0, ii=0.25)
```

dof (*x*)

Return the degrees-of-freedom

Returns inf when the degrees of freedom is greater than 1E6

Examples:

```
>>> ur = ureal(2.5,0.5,3,label='x')
>>> dof(ur)
3.0
```

(continues on next page)

(continued from previous page)

```
>>> ur.df
3.0

>>> uc = ucomplex(1+2j,(.3,.2),3,label='x')
>>> dof(uc)
3.0
```

label (*x*)
Return the label

component (*y*, *x*)
Return the magnitude of the component of uncertainty in `y` due to `x`.

Parameters

- **y** (*UncertainReal* or *UncertainComplex*) – an uncertain number
- **x** (*UncertainReal* or *UncertainComplex*) – an uncertain number

Return type float

If `x` and `y` are uncertain real, the function calls `reporting.u_component()` and returns the magnitude of the result.

If either `x` or `y` is uncertain complex, the returned value represents the magnitude of the component of uncertainty matrix (this is obtained by applying `reporting.u_bar()` to the result obtained from `reporting.u_component()`).

If either `x` or `y` is a number, zero is returned.

`component` can also e used in conjunction with `result()` to evaluate a component of uncertainty with respect to an intermediate uncertain number.

Examples:

```
>>> x1 = ureal(2,1)
>>> x2 = ureal(5,1)
>>> y = x1/x2
>>> reporting.u_component(y,x2)
-0.08
>>> component(y,x2)
0.08

>>> z1 = ucomplex(1+2j,1)
>>> z2 = ucomplex(3-2j,1)
>>> y = z1 - z2
>>> reporting.u_component(y,z2)
ComponentOfUncertainty(rr=-1.0, ri=0.0, ir=0.0, ii=-1.0)
>>> component(y,z2)
1.0

>>> I = ureal(1E-3,1E-5)
>>> R = ureal(1E3,1)
>>> V = result( I*R )
>>> P = V**2/R
>>> component(P,V)
2.0099751242241783e-05
```

get_covariance (*arg1*, *arg2=None*)
Evaluate covariance.

The input arguments can be a pair of uncertain numbers, or a single uncertain complex number.

When a pair of uncertain real numbers is supplied, the correlation between the two arguments is returned as a real number.

When one, or both, arguments are uncertain complex numbers, a *CovarianceMatrix* is returned, representing a 2-by-2 variance-covariance matrix.

get_correlation (*arg1*, *arg2=None*)
Return correlation

The input arguments may be a pair of uncertain numbers, or a single uncertain complex number.

When a pair of uncertain real numbers is provided, the correlation between the arguments is returned as a real number.

When one, or both, arguments are uncertain complex numbers, a *CorrelationMatrix* is returned, representing a 2-by-2 matrix of correlation coefficients.

set_correlation (*r*, *arg1*, *arg2=None*)
Set correlation between elementary uncertain numbers

The input arguments can be a pair of uncertain numbers (the same type, real or complex), or a single uncertain complex number.

The uncertain number arguments must be elementary uncertain numbers.

If the arguments have finite degrees of freedom, they must be declared together using either *multiple_ureal()* or *multiple_ucomplex()*.

If the uncertain number arguments have infinite degrees of freedom they can, alternatively, be declared by setting the argument *independent=False* when calling *ureal()* or *ucomplex()*.

A `ValueError` is raised when illegal arguments are used

When a pair of uncertain real numbers is provided, `r` is the correlation coefficient between them.

When a pair of uncertain complex number arguments is provided, `r` must be a 4-element sequence containing correlation coefficients between the components of the complex quantities.

Examples:

```
>>> x1 = ureal(2,1,independent=False)
>>> x2 = ureal(5,1,independent=False)
>>> set_correlation(.3,x1,x2)
>>> get_correlation(x1,x2)
0.3

>>> z = ucomplex(1+0j,(1,1),independent=False)
>>> z
ucomplex((1+0j), u=[1.0,1.0], r=0.0, df=inf)
>>> set_correlation(0.5,z)
>>> z
ucomplex((1+0j), u=[1.0,1.0], r=0.0, df=inf)

>>> x1 = ucomplex(1,(1,1),independent=False)
>>> x2 = ucomplex(1,(1,1),independent=False)
>>> correlation_mat = (0.25,0.5,0.75,0.5)
>>> set_correlation(correlation_mat,x1,x2)
>>> get_correlation(x1,x2)
CorrelationMatrix(rr=0.25, ri=0.5, ir=0.75, ii=0.5)
```

result (*un*, *label=None*)
Define an uncertain number as an intermediate result

Parameters

- **un** – an uncertain number or `UncertainArray`
- **label** – a string or sequence of strings

When `un` is an array, an `UncertainArray` is returned containing the intermediate uncertain number objects.

Note: This function is best applied to a temporary object, because a new intermediate result object is created. The original object `un` is not affected.

The component of uncertainty, or the sensitivity, of an uncertain number with respect to an intermediate result can be evaluated.

Declaring intermediate results also enables the dependencies of uncertain numbers to be stored in an archive.

Parameters

- **un** – *UncertainReal* or *UncertainComplex* or `UncertainArray`
- **label** – str or a sequence of str

Return type *UncertainReal* or *UncertainComplex* or `UncertainArray`

Example:

```
>>> I = ureal(1.3E-3,0.01E-3)
>>> R = ureal(995,7)
>>> V = result( I*R )
>>> P = V**2/R
>>> component(P,V)
3.505784505642068e-05
```

cos (*x*)
Uncertain number cosine function

sin (*x*)
Uncertain number sine function

tan (*x*)
Uncertain number tangent function

acos (*x*)
Uncertain number arc-cosine function

Note: In the complex case there are two branch cuts: one extends right, from 1 along the real axis to ∞, continuous from below; the other extends left, from -1 along the real axis to $-\infty$, continuous from above.

asin (*x*)
Uncertain number arcsine function

Note: In the complex case there are two branch cuts: one extends right, from 1 along the real axis to ∞, continuous from below; the other extends left, from -1 along the real axis to $-\infty$, continuous from above.

atan (*x*)
Uncertain number arctangent function

Note: In the complex case there are two branch cuts: One extends from j along the imaginary axis to $j\infty$, continuous from the right. The other extends from $-j$ along the imaginary axis to $-j\infty$, continuous from the left.

atan2 (*y*, *x*)
Two-argument uncertain number arctangent function

Parameters

- **x** (*UncertainReal*) – abscissa

- **y** (*UncertainReal*) – ordinate

Note: this function is not defined for uncertain complex numbers (use *phase()*)

Example:

```
>>> x = ureal(math.sqrt(3)/2,1)
>>> y = ureal(0.5,1)
>>> theta = atan2(y,x)
>>> theta
ureal(0.5235987755982989,1.0,inf)
>>> math.degrees( theta.x )
30.000000000000004
```

exp (*x*)
Uncertain number exponential function

pow (*x*, *y*)
Uncertain number power function

Raises x to the power of y

log (*x*)
Uncertain number natural logarithm

Note: In the complex case there is one branch cut, from 0 along the negative real axis to $-\infty$, continuous from above.

log10 (*x*)
Uncertain number common logarithm (base-10)

Note: In the complex case there is one branch cut, from 0 along the negative real axis to $-\infty$, continuous from above.

sqrt (*x*)
Uncertain number square root function

Note: In the complex case there is one branch cut, from 0 along the negative real axis to $-\infty$, continuous from above.

sinh (*x*)
Uncertain number hyperbolic sine function

cosh (*x*)
Uncertain number hyperbolic cosine function

tanh (*x*)
Uncertain number hyperbolic tangent function

acosh (*x*)
Uncertain number hyperbolic arc-cosine function

Note: In the complex case there is one branch cut, extending left from 1 along the real axis to $-\infty$, continuous from above.

asinh (*x*)
Uncertain number hyperbolic arcsine function

Note: In the complex case there are two branch cuts: one extends from j along the imaginary axis to j∞, continuous from the right; the other extends from −j along the imaginary axis to −j∞, continuous from the left.

atanh(*x*)
Uncertain number hyperbolic arctangent function

Note: In the complex case there are two branch cuts: one extends from 1 along the real axis to ∞, continuous from below; the other extends from -1 along the real axis to −∞, continuous from above.

mag_squared(*x*)
Return the squared magnitude of `x`.

Note: If `x` is an uncertain number, the magnitude squared is returned as an uncertain real number, otherwise :func:`abs(x)**2` is returned.

magnitude(*x*)
Return the magnitude of `x`

Note: If `x` is not an uncertain number type, returns `abs(x)`.

phase(*z*)

Parameters **z** (*UncertainComplex*) – an uncertain complex number

Returns the phase in radians

Return type *UncertainReal*

2.1.2 Uncertain Number Types

There are two types of uncertain number, one to represent real-valued quantities (*UncertainReal*) and one to represent real-complex quantities (*UncertainComplex*).

Uncertain Real Numbers

UncertainReal defines an uncertain-number object with attributes `x`, `u`, `v` and `df`, for the value, uncertainty, variance and degrees-of-freedom, respectively, of the uncertain number.

The function *ureal()* creates elementary *UncertainReal* objects. For example,

```
>>> x = ureal(1.414141,0.01)
>>> x
ureal(1.414141,0.01,inf)
```

All logical comparison operations (e.g., <, >, ==, etc) applied to uncertain-number objects use the *value* attribute. For example,

```
>>> un = ureal(2.5,1)
>>> un > 3
False
>>> un == 2.5
True
```

When the value of an `UncertainReal` is converted to a string (e.g., by `str`, or by `print()`), the precision displayed depends on the uncertainty. The two least significant digits of the value correspond to the two most significant digits of the standard uncertainty. The value of standard uncertainty is appended to the string between parentheses.

For example,

```
>>> x = ureal(1.414141,0.01)
>>> str(x)
'1.414(10)'
>>> print(x)
1.414(10)
```

When an `UncertainReal` is converted to its Python *representation* (e.g., by `repr()`) a string is returned that shows the representation of the elements that define the uncertain number.

For example,

```
>>> x = ureal(1.4/3,0.01,5,label='x')
>>> repr(x)
"ureal(0.4666666666666666,0.01,5.0, label='x')"
```

class UncertainReal(*x, u_comp, d_comp, i_comp, node=None*)

An `UncertainReal` holds information about the measured value of a real-valued quantity

conjugate()

Return the complex conjugate

Return type `UncertainReal`

df

Return the degrees of freedom

Return type float

Note `un.df` is equivalent to *`dof(un)`*

Example::

```
>>> ur = ureal(2.5,0.5,3)
>>> ur.df
3.0
```

imag

Returns the imaginary component

label

The uncertain-number label

Return type str

Note `un.label` is equivalent to *`label(un)`*

Example::

```
>>> x = ureal(2.5,0.5,label='x')
>>> x.label
'x'
```

```
>>> label(x)
'x'
```

real

Return the real component

u

Return the standard uncertainty

Return type float

Note that `un.u` is equivalent to *uncertainty(un)*

Example:

```
>>> ur = ureal(2.5,0.5)
>>> ur.u
0.5
```

v

Return the standard variance

Return type float

Note that `un.v` is equivalent to *variance(un)*

Example::

```
>>> ur = ureal(2.5,0.5)
>>> ur.v
0.25
```

x

Return the value

Return type float

Note that `un.x` is equivalent to *value(un)*

Example::

```
>>> ur = ureal(2.5,0.5)
>>> ur.x
2.5
```

Uncertain Complex Numbers

UncertainComplex defines an uncertain-number object with attributes `x`, `u`, `v` and `df`, for the value, uncertainty, variance-covariance matrix and degrees-of-freedom, respectively.

The function *ucomplex()* creates elementary *UncertainComplex* objects, for example

```
>>> z = ucomplex(1.333-0.121212j,(0.01,0.01))
```

Equality comparison operations (`==` and `!=`) applied to uncertain-complex-number objects use the *value* attribute. For example,

```
>>> uc = ucomplex(3+3j,(1,1))
>>> uc == 3+3j
True
```

The built-in function `abs()` returns the magnitude of the *value* as a Python `float` (use *magnitude()* if uncertainty propagation is required). For example,

```
>>> uc = ucomplex(1+1j,(1,1))
>>> abs(uc)
1.4142135623730951

>>> magnitude(uc)
ureal(1.4142135623730951,0.9999999999999999,inf)
```

When an *UncertainComplex* is converted to a string (e.g., by the `str` function or by `print()`), the precision depends on the uncertainty.

The lesser of the uncertainties in the real and imaginary components will determine the precision displayed. The two least significant digits of the formated component values will correspond to the two most significant digits of this standard uncertainty. Values of standard uncertainty are appended to the component values between parentheses.

For example,

```
>>> z = ucomplex(1.333-0.121212j,(0.01,0.002))
>>> print(z)
(1.3330(100)-0.1212(20)j)
```

When an `UncertainComplex` is converted to its Python *representation* (e.g., by `repr()`), a string is returned that shows the representation of the elements that define the uncertain number.

For example,

```
>>> z = ucomplex(1.333-0.121212j,(0.01,0.002))
>>> repr(z)
'ucomplex((1.333-0.121212j), u=[0.01,0.002], r=0.0, df=inf)'
```

class UncertainComplex(*r*, *i*)

An `UncertainComplex` holds information about the measured value of a complex-valued quantity

conjugate()

Return the complex conjugate

An `UncertainComplex` object is created by negating the imaginary component.

Return type `UncertainComplex`

df

Return the degrees-of-freedom

When the object is not an elementary uncertain number, the effective degrees-of-freedom is calculated using the method described by Willink and Hall in Metrologia 2002, 39, pp 361-369.

Return type float

Note that `uc.df` is equivalent to `dof(uc)`

Example::

```
>>> uc = ucomplex(1+2j,(.3,.2),3)
>>> uc.df
3.0
```

imag

The imaginary component.

Type `UncertainReal`

label

The *label* attribute

Return type str

Note that‘‘un.label‘‘ is equivalent to `label(un)`

Example::

```
>>> z = ucomplex(2.5+.3j,(1,1),label='z')
>>> z.label
'z'
```

r

Return the correlation coefficient between real and imaginary components

Return type float

real

The real component.

Type *UncertainReal*

u

Return standard uncertainties for the real and imaginary components

Return type 2-element sequence of float

Note that `uc.u` is equivalent to *uncertainty(uc)*

Example:

```
>>> uc = ucomplex(1+2j,(.5,.5))
>>> uc.u
StandardUncertainty(real=0.5, imag=0.5)
```

v

Return the variance-covariance matrix

The uncertainty of an uncertain complex number can be associated with a 4-element variance-covariance matrix.

Return type 4-element sequence of float

Note that `uc.v` is equivalent to *variance(uc)*

Example:

```
>>> uc = ucomplex(1+2j,(.5,.5))
>>> uc.v
VarianceCovariance(rr=0.25, ri=0.0, ir=0.0, ii=0.25)
```

x

Return the value

Return type complex

Note that `uc.x` is equivalent to *value(uc)*

Example::

```
>>> uc = ucomplex(1+2j,(.3,.2))
>>> uc.x
(1+2j)
```

2.2 Evaluating type-A uncertainty

A type-A evaluation of uncertainty involves statistical analysis of data (in contrast to a type-B evaluation, which by some means other than statistical analysis).

The shorter name `ta` has been defined as an alias for `type_a`, to resolve the names of objects defined in this module.

2.2.1 Sample estimates

- *estimate()* returns an uncertain number defined from the statistics of a sample of data.
- *multi_estimate_real()* returns a sequence of related uncertain real numbers defined from the multivariate statistics calculated from a sample of data.
- *multi_estimate_complex()* returns a sequence of related uncertain complex numbers defined from the multivariate statistics of a sample of data.

- `estimate_digitized()` returns an uncertain number for the mean of a sample of digitized data.
- `mean()` returns the mean of a sample of data.
- `standard_uncertainty()` evaluates the standard uncertainty associated with the sample mean.
- `standard_deviation()` evaluates the standard deviation of a sample of data.
- `variance_covariance_complex()` evaluates the variance and covariance associated with the mean real component and mean imaginary component of the data.

Note: Many functions in `type_a` treat data as pure numbers. Sequences of uncertain numbers can be passed to these functions, but only the uncertain-number values will be used.

2.2.2 Module contents

estimate(*seq*, *label=None*, *context=<GTC.context.Context object>*)

Return an uncertain number for the mean of the data

Parameters

- **seq** – a sequence of data
- **label** (`str`) – a label for the returned uncertain number

Return type `UncertainReal` or `UncertainComplex`

The elements of `seq` may be real numbers, complex numbers, or uncertain real or complex numbers. Note that only the value of uncertain numbers will be used.

In a type-A evaluation, the sample mean provides an estimate of the quantity of interest. The uncertainty in this estimate is the standard deviation of the sample mean (or the sample covariance of the mean, in the complex case).

The function returns an `UncertainReal` when the mean of the data is real, and an `UncertainComplex` when the mean of the data is complex.

Examples:

```
>>> data = range(15)
>>> type_a.estimate(data)
ureal(7.0,1.1547005383792515,14)

>>> data = [(0.91518731126816899+1.5213442955575518j),
... (0.96572684493613492-0.18547192979059401j),
... (0.23216598132006649+1.6951311687588568j),
... (2.1642786101267397+2.2024333895672563j),
... (1.1812532664590505+0.59062101107787357j),
... (1.2259264339405165+1.1499373179910186j),
... (-0.99422341300318684+1.7359338393131392j),
... (1.2122867690240853+0.32535154897909946j),
... (2.0122536479379196-0.23283009302603963j),
... (1.6770229536619197+0.77195994890476838j)]

>>> type_a.estimate(data)
ucomplex((1.059187840567141+0.9574410497332932j), u=[0.28881665310241805,0.
↪2655555630050262], r=-4.090655272692547, df=9)
```

estimate_digitized(*seq*, *delta*, *label=None*, *truncate=False*, *context=<GTC.context.Context object>*)

Return an uncertain number for the mean of digitized data

Parameters

- **seq** (float, `UncertainReal` or `UncertainComplex`) – data

- **delta** (*float*) – digitization step size
- **label** (*str*) – label for uncertain number returned
- **truncate** (*bool*) – if `True`, truncation, rather than rounding, is assumed

Return type *UncertainReal* or *UncertainComplex*

A sequence of data that has been formatted with fixed precision can completely conceal a small amount of variability in the original values, or merely obscure that variability.

This function recognises the possible interaction between truncation, or rounding, errors and random errors in the underlying data. The function obtains the mean of the data sequence and evaluates the uncertainty in this mean as an estimate of the mean of the process generating the data.

Set the argument `truncate` to `True` if data have been truncated, instead of rounded.

See reference: R Willink, *Metrologia*, **44** (2007) 73-81

Examples:

```
# LSD = 0.0001, data varies between -0.0055 and -0.0057
>>> seq = (-0.0056,-0.0055,-0.0056,-0.0056,-0.0056,
...       -0.0057,-0.0057,-0.0056,-0.0056,-0.0057,-0.0057)
>>> type_a.estimate_digitized(seq,0.0001)
ureal(-0.005627272727272727,1.9497827808661157e-05,10)

# LSD = 0.0001, data varies between -0.0056 and -0.0057
>>> seq = (-0.0056,-0.0056,-0.0056,-0.0056,-0.0056,
... -0.0057,-0.0057,-0.0056,-0.0056,-0.0057,-0.0057)
>>> type_a.estimate_digitized(seq,0.0001)
ureal(-0.005636363636363636,1.5212000482437775e-05,10)

# LSD = 0.0001, no spread in data values
>>> seq = (-0.0056,-0.0056,-0.0056,-0.0056,-0.0056,
... -0.0056,-0.0056,-0.0056,-0.0056,-0.0056,-0.0056)
>>> type_a.estimate_digitized(seq,0.0001)
ureal(-0.0056,2.886751345948129e-05,10)

# LSD = 0.0001, no spread in data values, fewer points
>>> seq = (-0.0056,-0.0056,-0.0056)
>>> type_a.estimate_digitized(seq,0.0001)
ureal(-0.0056,3.291402943021917e-05,2)
```

multi_estimate_real (*seq_of_seq*, *labels=None*)

Return a sequence of uncertain real numbers

Parameters

- **seq_of_seq** – a sequence of sequences of data
- **labels** – a sequence of *str* labels

Return type seq of *UncertainReal*

The sequences in `seq_of_seq` must all be the same length. Each sequence is associated with a particular quantity and contains a sample of data. An uncertain number for the quantity will be created using the sample of data, using sample statistics. The covariance between different quantities will also be evaluated from the data.

A sequence of elementary uncertain numbers are returned. The uncertain numbers are considered related, allowing a degrees-of-freedom calculations to be performed on derived quantities.

Example:

```
# From Appendix H2 in the GUM
```

(continues on next page)

(continued from previous page)

```
>>> V = [5.007,4.994,5.005,4.990,4.999]
>>> I = [19.663E-3,19.639E-3,19.640E-3,19.685E-3,19.678E-3]
>>> phi = [1.0456,1.0438,1.0468,1.0428,1.0433]
>>> v,i,p = type_a.multi_estimate_real((V,I,phi),labels=('V','I','phi'))
>>> v
ureal(4.999,0.0032093613071761794,4, label='V')
>>> i
ureal(0.019661,9.471008394041335e-06,4, label='I')
>>> p
ureal(1.04446,0.0007520638270785368,4, label='phi')

>>> r = v/i*cos(p)
>>> r
ureal(127.732169928102...,0.071071407396995...,4.0)
```

multi_estimate_complex(*seq_of_seq*, *labels=None*, *context=<GTC.context.Context object>*)

Return a sequence of uncertain complex numbers

Parameters

- **seq_of_seq** – a sequence of sequences of data
- **labels** – a sequence of *str* labels

Return type a sequence of *UncertainComplex*

The sequences in `seq_of_seq` must all be the same length. Each sequence contains a sample of data that is associated with a particular quantity. An uncertain number for the quantity will be created using this data from sample statistics. The covariance between different quantities will also be evaluated from the data.

A sequence of elementary uncertain complex numbers are returned. These uncertain numbers are considered related, allowing a degrees-of-freedom calculations to be performed on derived quantities.

Defines uncertain numbers using the sample statistics, including the sample covariance.

Example:

```
# From Appendix H2 in the GUM

>>> I = [ complex(x) for x in (19.663E-3,19.639E-3,19.640E-3,19.685E-3,19.678E-
↪3) ]
>>> V = [ complex(x) for x in (5.007,4.994,5.005,4.990,4.999)]
>>> P = [ complex(0,p) for p in (1.0456,1.0438,1.0468,1.0428,1.0433) ]

>>> v,i,p = type_a.multi_estimate_complex( (V,I,P) )

>>> get_correlation(v.real,i.real)
-0.355311219817512

>>> z = v/i*exp(p)
>>> z.real
ureal(127.732169928102...,0.071071407396995...,4.0)
>>> get_correlation(z.real,z.imag)
-0.588429784423515...
```

mean(*seq*, **args*, ***kwargs*)

Return the arithmetic mean of data in `seq`

Parameters

- **seq** – a sequence, `ndarray`, or iterable, of numbers or uncertain numbers
- **args** – optional arguments when `seq` is an `ndarray`
- **kwargs** – optional keyword arguments when `seq` is an `ndarray`

If `seq` contains real or uncertain real numbers, a real number is returned.

If `seq` contains complex or uncertain complex numbers, a complex number is returned.

Example:

```
>>> data = range(15)
>>> type_a.mean(data)
7.0
```

standard_deviation(*seq*, *mu=None*)

Return the sample standard deviation

Parameters

- **seq** – sequence of data
- **mu** – the arithmetic mean of `seq`

If `seq` contains real or uncertain real numbers, the sample standard deviation is returned.

If `seq` contains complex or uncertain complex numbers, the standard deviation in the real and imaginary components is evaluated, as well as the correlation coefficient between the components. The results are returned in a pair of objects: a *StandardDeviation* namedtuple and a correlation coefficient.

Only the values of uncertain numbers are used in calculations.

Examples:

```
>>> data = range(15)
>>> type_a.standard_deviation(data)
4.47213595499958

>>> data = [(0.91518731126816899+1.5213442955575518j),
... (0.96572684493613492-0.18547192979059401j),
... (0.23216598132006649+1.6951311687588568j),
... (2.1642786101267397+2.2024333895672563j),
... (1.1812532664590505+0.59062101107787357j),
... (1.2259264339405165+1.1499373179910186j),
... (-0.99422341300318684+1.7359338393131392j),
... (1.2122867690240853+0.32535154897909946j),
... (2.0122536479379196-0.23283009302603963j),
... (1.6770229536619197+0.77195994890476838j)]
>>> sd,r = type_a.standard_deviation(data)
>>> sd
StandardDeviation(real=0.913318449990377, imag=0.8397604244242309)
>>> r
-0.31374045124595246
```

standard_uncertainty(*seq*, *mu=None*)

Return the standard uncertainty of the sample mean

Parameters

- **seq** – sequence of data
- **mu** – the arithmetic mean of `seq`

Return type float or *StandardUncertainty*

If `seq` contains real or uncertain real numbers, the standard uncertainty of the sample mean is returned.

If `seq` contains complex or uncertain complex numbers, the standard uncertainties of the real and imaginary components are evaluated, as well as the sample correlation coefficient are returned in a *StandardUncertainty* namedtuple

Only the values of uncertain numbers are used in calculations.

Example:

```
>>> data = range(15)
>>> type_a.standard_uncertainty(data)
1.1547005383792515

>>> data = [(0.91518731126816899+1.5213442955575518j),
... (0.96572684493613492-0.18547192979059401j),
... (0.23216598132006649+1.6951311687588568j),
... (2.1642786101267397+2.2024333895672563j),
... (1.1812532664590505+0.59062101107787357j),
... (1.2259264339405165+1.1499373179910186j),
... (-0.99422341300318684+1.7359338393131392j),
... (1.2122867690240853+0.32535154897909946j),
... (2.0122536479379196-0.23283009302603963j),
... (1.6770229536619197+0.77195994890476838j)]
>>> u,r = type_a.standard_uncertainty(data)
>>> u
StandardUncertainty(real=0.28881665310241805, imag=0.2655555630050262)
>>> u.real
0.28881665310241805
>>> r
-0.31374045124595246
```

variance_covariance_complex(*seq*, *mu=None*)

Return the sample variance-covariance matrix

Parameters

- **seq** – sequence of data
- **mu** – the arithmetic mean of `seq`

Returns a 4-element sequence

If `mu` is `None` the mean will be evaluated by *mean()*.

`seq` may contain numbers or uncertain numbers. Only the values of uncertain numbers are used in calculations.

Variance-covariance matrix elements are returned in a *VarianceCovariance* namedtuple; they can be accessed using the attributes `.rr`, `.ri`, `,ir` and `.ii`.

Example:

```
>>> data = [(0.91518731126816899+1.5213442955575518j),
... (0.96572684493613492-0.18547192979059401j),
... (0.23216598132006649+1.6951311687588568j),
... (2.1642786101267397+2.2024333895672563j),
... (1.1812532664590505+0.59062101107787357j),
... (1.2259264339405165+1.1499373179910186j),
... (-0.99422341300318684+1.7359338393131392j),
... (1.2122867690240853+0.32535154897909946j),
... (2.0122536479379196-0.23283009302603963j),
... (1.6770229536619197+0.77195994890476838j)]
>>> type_a.variance_covariance_complex(data)
VarianceCovariance(rr=0.8341505910928249, ri=-0.24062910264062262, ir=-0.
↪24062910264062262, ii=0.7051975704291644)

>>> v = type_a.variance_covariance_complex(data)
>>> v[0]
0.8341505910928249
>>> v.rr
0.8341505910928249
>>> v.ii
0.7051975704291644
```

2.3 Evaluating type-B uncertainty

The shorter name `tb` has been defined as an alias for *type_b*, to resolve the names of objects in this module.

2.3.1 Real-valued problems

Functions are provided that convert the half-width of a one-dimensional distribution to a standard uncertainty:

- *uniform()*
- *triangular()*
- *u_shaped()*
- *arcsine()*

2.3.2 Complex-valued problems

The following functions convert information about two-dimensional distributions into standard uncertainties:

- *uniform_ring()*
- *uniform_disk()*
- *unknown_phase_product()*

2.3.3 A table of distributions

The mapping `distribution` is provided so that the functions above can be selected by name. For example,

```
>>> a = 1.5
>>> ureal( 1, type_b.distribution['gaussian'](a) )
ureal(1.0,1.5,inf)
>>> ureal( 1, type_b.distribution['uniform'](a) )
ureal(1.0,0.8660254037844387,inf)
>>> ureal( 1, type_b.distribution['arcsine'](a) )
ureal(1.0,1.0606601717798212,inf)
```

Keys to `distribution` are (case-sensitive):

- *gaussian*
- *uniform*
- *triangular*
- *arcsine*
- *u_shaped*
- *uniform_ring*
- *uniform_disk*

2.3.4 Module contents

uniform(*a*)

Return the standard uncertainty for a uniform distribution.

Parameters a (*float*) – the half-width

Example:

```
>>> x = ureal(1,type_b.uniform(1))
>>> x
ureal(1.0,0.5773502691896258,inf)
```

triangular(*a*)

Return the standard uncertainty for a triangular distribution.

Parameters a (*float*) – the half-width

Example:

```
>>> x = ureal(1,type_b.triangular(1))
>>> x
ureal(1.0,0.4082482904638631,inf)
```

u_shaped(*a*)

Return the standard uncertainty for an arcsine distribution.

Parameters a (*float*) – the half-width

Example:

```
>>> x = ureal(1,type_b.arcsine(1))
>>> x
ureal(1.0,0.7071067811865475,inf)
```

arcsine(*a*)

Return the standard uncertainty for an arcsine distribution.

Parameters a (*float*) – the half-width

Example:

```
>>> x = ureal(1,type_b.arcsine(1))
>>> x
ureal(1.0,0.7071067811865475,inf)
```

uniform_ring(*a*)

Return the standard uncertainty for a uniform ring

Parameters a (*float*) – the radius

Convert the radius of a uniform ring distribution `a` to a standard uncertainty

See reference: B D Hall, *Metrologia* **48** (2011) 324-332

Example:

```
>>> z = ucomplex( 0, type_b.uniform_ring(1) )
>>> z
ucomplex((0+0j), u=[0.7071067811865475,0.7071067811865475], r=0.0, df=inf)
```

uniform_disk(*a*)

Return the standard uncertainty for a uniform disk

Parameters a (*float*) – the radius

Convert the radius of a uniform disk distribution `a` to a standard uncertainty.

See reference: B D Hall, *Metrologia* **48** (2011) 324-332

Example:

```
>>> z = ucomplex( 0, type_b.uniform_disk(1) )
>>> z
ucomplex((0+0j), u=[0.5,0.5], r=0.0, df=inf)
```

unknown_phase_product(*u1*, *u2*)

Return the standard uncertainty for a product when phases are unknown

Parameters

- **u1** – the standard uncertainty of the first multiplicand
- **u2** – the standard uncertainty of the second multiplicand

Obtains the standard uncertainty associated with a complex product when estimates have unknown phase.

The arguments `u1` and `u2` are the standard uncertainties associated with each multiplicand.

See reference: B D Hall, *Metrologia* **48** (2011) 324-332

Example:

```
# X = Gamma1 * Gamma2
>>> X = ucomplex( 0, type_b.unknown_phase_product(.1,.1) )
>>> X
ucomplex((0+0j), u=[0.014142135623730954,0.014142135623730954], r=0.0, df=inf)
```

2.4 function module

2.4.1 Utility functions

Functions *complex_to_seq()* and *seq_to_complex()* are useful to convert between the matrix representation of complex numbers and Python `complex`.

The function *mean()* evaluates the mean of a sequence.

2.4.2 Module contents

complex_to_seq(*z*)

Transform a complex number into a 4-element sequence

Parameters **z** – a number

If `z = x + yj`, then an array of the form `[[x,-y],[y,x]]` can be used to represent `z` in matrix computations.

Examples::

```
>>> import numpy
>>> z = 1 + 2j
>>> function.complex_to_seq(z)
(1.0, -2.0, 2.0, 1.0)
```

```
>>> m = numpy.array( function.complex_to_seq(z) )
>>> m.shape = (2,2)
>>> print( m )
[[ 1. -2.]
 [ 2.  1.]]
```

seq_to_complex (*seq*)

Transform a 4-element sequence into a complex number

Parameters **seq** – a 4-element sequence

Raises RuntimeError – if seq is ill-conditioned

If z = x + yj, then an array of the form [[x,-y],[y,x]] can be used to represent z in matrix computations.

Examples:

```
>>> import numpy
>>> seq = (1,-2,2,1)
>>> z = function.seq_to_complex( seq )
>>> z
(1+2j)
>>> a = numpy.array((1,-2,2,1))
>>> a.shape = 2,2
>>> a
array([[ 1, -2],
       [ 2,  1]])
>>> z = function.seq_to_complex(a)
>>> z
(1+2j)
```

mean (*seq*, **args*, ***kwargs*)

Return the arithmetic mean of the elements in *seq*

Parameters

- **seq** – a sequence, ndarray, or iterable, of numbers or uncertain numbers
- **args** – optional arguments when seq is an ndarray
- **kwargs** – optional keyword arguments when seq is an ndarray

If the elements of seq are uncertain numbers, an uncertain number is returned.

Example

```
>>> seq = [ ureal(1,1), ureal(2,1), ureal(3,1) ]
>>> function.mean(seq)
ureal(2.0,0.5773502691896257,inf)
```

2.5 Reporting functions

This module provides functions to facilitate the reporting of information about calculations.

The shorter name rp has been defined as an alias for *reporting*, to resolve the names of objects defined in this module.

2.5.1 Reporting functions

- The function *budget()* produces an uncertainty budget.
- The function *k_factor()* returns the coverage factor used for real-valued problems (based on the Student-t distribution).
- The function *k_to_dof()* returns the degrees of freedom corresponding to a given coverage factor and coverage probability.
- The function *k2_factor_sq()* returns coverage factor squared for the complex-valued problem.

- The function *k2_to_dof()* returns the degrees of freedom corresponding to a given coverage factor and coverage probability in complex-valued problems.
- Functions *u_bar()* and *v_bar()* return summary values for matrix results associated with 2-D uncertainty.

2.5.2 Uncertainty functions

- The function *u_component()* returns the signed component of uncertainty in one uncertain number due to uncertainty in another.
- The function *sensitivity()* returns the partial derivative of one uncertain number with respect to another. This is ofetn called the sensitivity coefficient.

2.5.3 Type functions

- The function *is_ureal()* can be used to identify uncertain real numbers.
- The function *is_ucomplex()* can be used to identify uncertain complex numbers.

2.5.4 Module contents

budget (*y*, *influences=None*, *key='u'*, *reverse=True*, *trim=0.01*, *max_number=None*)

Return a sequence of label-component of uncertainty pairs

Parameters

- **y** (*UncertainReal* or *UncertainComplex*) – an uncertain number
- **influences** – a sequence of uncertain numbers
- **key** – the list sorting key
- **reverse** (*bool*) – determines sorting order (forward or reverse)
- **trim** – remove components of uncertainty that are less than `trim` times the largest component
- **max_number** – return no more than `max_number` components

A sequence of *Influence* namedtuples is returned, each with the attributes `label` and `u` for a component of uncertainty (see *component()*).

The argument `influences` can be used to select the influences are that reported.

The argument `key` can be used to order the sequence by the component of uncertainty or the label (`u` or `label`).

The argument `reverse` controls the sense of ordering.

The argument `trim` can be used to set a minimum relative magnitude of components returned. Set `trim=0` for a complete list.

The argument `max_number` can be used to restrict the number of components returned.

Example:

```
>>> x1 = ureal(1,1,label='x1')
>>> x2 = ureal(2,0.5,label='x2')
>>> x3 = ureal(3,0.1,label='x3')
>>> y = (x1 - x2) / x3
>>> for l,u in reporting.budget(y):
...     print("{0}: {1:G}".format(l,u))
...
```

(continues on next page)

(continued from previous page)

```
x1: 0.333333
x2: 0.166667
x3: 0.0111111

>>> for l,u in reporting.budget(y,reverse=False):
...     print("{0}: {1:G}".format(l,u))
...
x3: 0.0111111
x2: 0.166667
x1: 0.333333
```

k_factor (*df=inf*, *p=95*)

Return the a coverage factor for an uncertainty interval

Parameters

- **df** (*float*) – the degrees-of-freedom (>1)
- **p** (*int or float*) – the coverage probability (%)

Evaluates the coverage factor for an uncertainty interval with coverage probability `p` and degrees-of-freedom `df` based on the Student t-distribution.

Example:

```
>>> reporting.k_factor(3)
3.182446305284263
```

k_to_dof (*k*, *p=95*)

Return the dof corresponding to a univariate coverage factor *k*

Parameters

- **k** (*float*) – coverage factor (>0)
- **p** (*int or float*) – coverage probability (%)

Evaluates the degrees-of-freedom given a coverage factor for an uncertainty interval with coverage probability `p` based on the Student t-distribution.

Example:

```
>>> reporting.k_to_dof(2.0,95)
60.43756442698591
```

k2_factor_sq (*df=inf*, *p=95*)

Return a squared coverage factor for an elliptical uncertainty region

Parameters

- **df** (*float*) – the degrees-of-freedom (>=2)
- **p** (*int or float*) – the coverage probability (%)

Evaluates the square of the coverage factor for an elliptical uncertainty region with coverage probability `p` and `df` degrees of freedom based on the F-distribution.

Example:

```
>>> reporting.k2_factor_sq(3)
    56.99999999999994
```

k2_to_dof (*k2*, *p=95*)

Return the dof corresponding to a bivariate coverage factor *k2*

Parameters

- **k2** (*float*) – coverage factor (>0)
- **p** (*int or float*) – coverage probability (%)

Evaluates a number of degrees-of-freedom given a coverage factor for an elliptical uncertainty region with coverage probability p based on the F-distribution.

Example:

```
>>> reporting.k2_to_dof(2.6,95)
34.35788424389927
```

u_component (*y*, *x*)

Return the component of uncertainty in y due to x

Parameters

- **y** – *UncertainReal* or *UncertainComplex* or UncertainArray
- **x** – *UncertainReal* or *UncertainComplex* or UncertainArray

If x and y are uncertain real numbers, return a float.

If y or x is an uncertain complex number, return a 4-element sequence of float, containing the components of uncertainty.

When x and y are arrays, an *uncertain_array.UncertainArray* is returned containing the results of applying this function to the array elements.

Otherwise, return 0.

Example:

```
>>> x = ureal(3,1)
>>> y = 3 * x
>>> reporting.u_component(y,x)
3.0

>>> q = ucomplex(2,1)
>>> z = magnitude(q)    # uncertain real numbers
>>> reporting.u_component(z,q)
ComponentOfUncertainty(rr=1.0, ri=0.0, ir=0.0, ii=0.0)

>>> r = ucomplex(3,1)
>>> z = q * r
>>> reporting.u_component(z,q)
ComponentOfUncertainty(rr=3.0, ri=-0.0, ir=0.0, ii=3.0)
```

sensitivity (*y*, *x*)

Return the first partial derivative of y with respect to x

Parameters

- **y** – *UncertainReal* or *UncertainComplex* or UncertainArray
- **x** – *UncertainReal* or *UncertainComplex* or UncertainArray

If x and y are uncertain real numbers, return a float.

If y or x is an uncertain complex number, return a 4-element sequence of float, representing the Jacobian matrix.

When x and y are arrays, an UncertainArray is returned containing the results of applying this function to the array elements.

Otherwise, return 0.

New in version 1.1.

Example:

```
>>> x = ureal(3,1)
>>> y = 3 * x
>>> reporting.sensitivity(y,x)
3.0

>>> q = ucomplex(2,1)
>>> z = magnitude(q)     # uncertain real numbers
>>> reporting.sensitivity(z,q)
JacobianMatrix(rr=1.0, ri=0.0, ir=0.0, ii=0.0)

>>> r = ucomplex(3,1)
>>> z = q * r
>>> reporting.sensitivity(z,q)
JacobianMatrix(rr=3.0, ri=-0.0, ir=0.0, ii=3.0)
```

is_ureal(*x*)

Return `True` if `x` is an uncertain real number

Example:

```
>>> x = ureal(1,1)
>>> reporting.is_ureal(x)
True
```

is_ucomplex(*z*)

Return `True` if `z` is an uncertain complex number

Example:

```
>>> z = ucomplex(1+2j,(0.1,0.2))
>>> reporting.is_ucomplex(z)
True
```

v_bar(*cv*)

Return the trace of `cv` divided by 2

Parameters **cv** (*4-element sequence of float*) – a variance-covariance matrix

Returns float

Example:

```
>>> x1 = 1-.5j
>>> x2 = .2+7.1j
>>> z1 = ucomplex(x1,(1,.2))
>>> z2 = ucomplex(x2,(.2,1))
>>> y = z1 * z2
>>> y.v
VarianceCovariance(rr=2.3464, ri=1.8432, ir=1.8432, ii=51.4216)
>>> reporting.v_bar(y.v)
26.884
```

u_bar(*ucpt*)

Return the magnitude of a component of uncertainty

Parameters **ucpt** (*float or 4-element sequence of float*) – a component of uncertainty

If `ucpt` is a sequence, return the root-sum-square of the elements divided by $\sqrt{2}$

If `ucpt` is a number, return the absolute value.

Example:

```
>>> x1 = 1-.5j
>>> x2 = .2+7.1j
>>> z1 = ucomplex(x1,1)
>>> z2 = ucomplex(x2,1)
>>> y = z1 * z2
>>> dy_dz1 = reporting.u_component(y,z1)
>>> dy_dz1
ComponentOfUncertainty(rr=0.2, ri=-7.1, ir=7.1, ii=0.2)
>>> reporting.u_bar(dy_dz1)
7.102816342831905
```

2.6 The persistence module

2.6.1 Class

An *Archive* object can be used to marshal a set of uncertain numbers for storage, or restore a set of uncertain numbers from storage.

Python pickle is used for the storage mechanism.

2.6.2 Functions

An archive can be pickled and stored in a file, or a string.

Functions for storing and retrieving a pickled archive file are

- *load()*
- *dump()*

Functions for storing and retrieving a pickled archive string are

- *dumps()*
- *loads()*

2.6.3 Module contents

class Archive

An *Archive* object can be used to marshal a set of uncertain numbers for storage, or restore a set of uncertain numbers from storage.

__getitem__(*key*)

Extract an uncertain number

key - the name of the archived number

__len__()

Return the number of entries

__setitem__(*key*, *value*)

Add an uncertain number to the archive

Example:

```
>>> a = Archive()
>>> x = ureal(1,1)
>>> y = ureal(2,1)
>>> a['x'] = x
>>> a['fred'] = y
```

add(***kwargs*)
Add entries `name = uncertain-number` to the archive

Example:

```
>>> a = Archive()
>>> x = ureal(1,1)
>>> y = ureal(2,1)
>>> a.add(x=x,fred=y)
```

extract(**args*)
Extract one or more uncertain numbers

Parameters **args** – names of archived uncertain numbers

If just one name is given, a single uncertain number is returned, otherwise a sequence of uncertain numbers is returned.

Example:

```
# >>> x, fred = a.extract('x','fred')
# >>> harry = a.extract('harry')
```

items()
Return a list of name -to- uncertain-number pairs

iteritems()
Return an iterator of name -to- uncertain-number pairs

iterkeys()
Return an iterator for names

itervalues()
Return an iterator for uncertain numbers

keys()
Return a list of names

values()
Return a list of uncertain numbers

load(*file*)
Load an archive from a file

Parameters **file** – a file object opened in binary read mode (with 'rb')

Several archives can be extracted from one file by repeatedly calling this function.

dump(*file*, *ar*)
Save an archive in a file

Parameters

- **file** – a file object opened in binary write mode (with 'wb')
- **ar** – an `Archive` object

Several archives can be saved in a file by repeated use of this function.

dumps(*ar*, *protocol=4*)
Return a string representation of the archive

Parameters

- **ar** – an `Archive` object
- **protocol** – encoding type

Possible values for `protocol` are described in the Python documentation for the 'pickle' module.

`protocol=0` creates an ASCII string, but note that many (special) linefeed characters are embedded.

loads (*s*)
Return an archive object restored from a string representation

Parameters **s** – a string created by `dumps()`

2.7 named-tuples

A number of `namedtuple` class are used in `GTC` to return the results of calculations.

class VarianceAndDof (*cv*, *df*)
`namedtuple`: Values of the variance and degrees of freedom.

cv
Variance.

df
`float`: Degrees of freedom.

class VarianceCovariance (*rr*, *ri*, *ir*, *ii*)
`namedtuple`: Values of variance-covariance for a complex quantity

rr
`float`: variance in the real component

ri
`float`: covariance between th real and imaginary components

ir
`float`: covariance between th real and imaginary components

ii
`float`: variance in the imaginary component

class StandardUncertainty (*real*, *imag*)
`namedtuple`: Standard uncertainty values of a complex quantity

real
`float`: standard uncertainty in the real component

imag
`float`: standard uncertainty in the imaginary component

class StandardDeviation (*real*, *imag*)
`namedtuple`: Standard deviation values of a complex quantity

real
`float`: standard deviation in the real component

imag
`float`: standard deviation in the imaginary component

class ComponentOfUncertainty (*rr*, *ri*, *ir*, *ii*)
`namedtuple`: Component of uncertainty values for a complex quantity

rr
`float`: real component with respect to real component

ri
`float`: real component with respect to imaginary component

ir
`float`: imaginary component with respect to real component

ii
`float`: imaginary component with respect to imaginary component

class Influence(*label*, *u*)
namedtuple: label and value of a component of uncertainty

label
str: influence quantity label

u
float: component of uncertainty due to influence quantity

class CovarianceMatrix(*rr*, *ri*, *ir*, *ii*)
namedtuple: Values of covariance for a pair of quantities `x` and `y`

rr
float: covariance between `x.real` and `y.real`

ri
float: covariance between `x.real` and `y.imag`

ir
float: covariance between `x.imag` and `y.real`

ii
float: covariance between `x.imag` and `y.imag`

class CorrelationMatrix(*rr*, *ri*, *ir*, *ii*)
namedtuple: Correlation coefficients for a pair of quantities `x` and `y`

rr
float: correlation between `x.real` and `y.real`

ri
float: correlation between `x.real` and `y.imag`

ir
float: correlation between `x.imag` and `y.real`

ii
float: correlation between `x.imag` and `y.imag`

2.8 Linear Algebra

This module provides support for calculations using arrays containing uncertain numbers.

The shorter name `la` has been defined as an alias for `linear_algebra`, to resolve the names of objects defined in this module.

2.8.1 Arrays of Uncertain Numbers

`UncertainArray` is a convenient container of uncertain numbers. The preferred way to create arrays is the function *uarray()*.

An array can contain a mixture of *UncertainReal*, *UncertainComplex* and Python numbers (`int`, `float` and `complex`).

The usual mathematical operations can be applied to an array. For instance, if `A` and `B` have the same size, they can be added `A + B`, subtracted `A - B`, etc; or a function like `sqrt(A)` can be applied. This vectorisation provides a succinct notation for repetitive operations but it does not offer a significant speed advantage over Python iteration.

Note: To evaluate the product of two-dimensional arrays representing matrices, the function *matmul()* should be used (for Python 3.5 and above the built-in binary operator `@` is an alternative). For example:

```
>>> a = la.uarray([[1.1,.5],[ureal(3,1),.5]])
>>> b = la.uarray([[5.2,ucomplex(4,1)],[.1,.1+3j]])
>>> la.matmul(a,b)
uarray([[5.7700000000000005,
         ucomplex((4.45+1.5j), u=[1.1,1.1], r=0.0, df=inf)],
        [ureal(15.650000000000002,5.2,inf),
         ucomplex((12.05+1.5j), u=[5.0,3.0], r=0.0, df=inf)]])
```

New in version 1.1.

Classes

- UncertainArray

Arithmetic operations

Arithmetic operations are defined for arrays (unary + and −, and binary +, − and *). The multiplication operator * is implemented element-wise. For two-dimensional arrays, matrix multiplication is performed by `matmul()` (since Python 3.5, the @ operator can be used). Also, `dot()` evaluates the array dot product, which for two-dimensional arrays is equivalent to matrix multiplication.

When one argument is a scalar, it is applied to each element of the array in turn.

Mathematical operations

The standard mathematical operations defined in `core` can be applied directly to an UncertainArray. An UncertainArray is returned, containing the result of the function applied to each element.

Functions

The functions `inv()`, `transpose()`, `solve()` and `det()` implement the usual linear algebra operations.

The functions `identity()`, `empty()`, `zeros()` `full()` and `ones()` create simple arrays.

Reporting functions

Reporting functions `u_component()` and `sensitivity()` can be applied directly to a pair of arrays. An UncertainArray containing the result of applying the function to pairs of elements will be returned.

The core *GTC* function `result()` can be used to define elements of an array as intermediate uncertain numbers.

Array broadcasting

When binary arithmetic operations are applied to arrays, the shape of the array may be changed for the purposes of the calculation. The rules are as follows:

- If arrays do not have the same number of dimensions, then dimensions of size *1* are prepended to the smaller array's shape

Following this, the size of array dimensions are compared and checked for compatibility. Array dimensions are compatible when

- dimension sizes are equal, or
- one of the dimension sizes is *1*

Finally, if either of the compared dimension sizes is *1*, the size of the larger dimension is used. For example:

```
>>> x = la.uarray([1,2])
>>> y = la.uarray([[1],[2]])
>>> print(x.shape,y.shape)
(2,) (2, 1)
>>> x + y
uarray([[2, 3],
        [3, 4]])
```

Module contents

uarray (*array*, *label=None*, *names=None*)
Create an array of uncertain numbers.

For an overview on how to use an `UncertainArray` see *Examples using UncertainArray*.

> **Attention:** Requires numpy ≥ v1.13.0 to be installed.

Parameters

- **array** – An array-like object containing `int`, `float`, `complex` *UncertainReal* or *UncertainComplex* elements.
- **label** (*str*) – A label to assign to the *array*. This *label* does not change labels previously assigned to array elements.
- **names** (*list[str]*) – The field *names* to use to create a structured array.

Returns An `UncertainArray`.

Examples:

Create an *amps* and a *volts* array and then calculate the *resistances*

```
>>> amps = la.uarray([ureal(0.57, 0.18), ureal(0.45, 0.12), ureal(0.68,
↪ 0.19)])
>>> volts = la.uarray([ureal(10.3, 1.3), ureal(9.5, 0.8), ureal(12.6,␣
↪1.9)])
>>> resistances = volts / amps
>>> resistances
uarray([ureal(18.070175438596493,6.145264246839438,inf),
        ureal(21.11111111111111,5.903661880050747,inf),
        ureal(18.52941176470588,5.883187720636909,inf)])
```

Create a Structured array, with the names `'amps'` and `'volts'`, and then calculate the *resistances*.

```
>>> data = la.uarray([(ureal(0.57, 0.18), ureal(10.3, 1.3)),
...                   (ureal(0.45, 0.12), ureal(9.5, 0.8)),
...                   (ureal(0.68, 0.19), ureal(12.6, 1.9))], names=['amps
↪', 'volts'])
>>> resistances = data['volts'] / data['amps']
>>> resistances
uarray([ureal(18.070175438596493,6.145264246839438,inf),
        ureal(21.11111111111111,5.903661880050747,inf),
        ureal(18.52941176470588,5.883187720636909,inf)])
```

dot (*lhs*, *rhs*)

Dot product of two arrays.

For more details see `numpy.dot()`.

Parameters

- **lhs** – The array-like object on the left-hand side.
- **rhs** – The array-like object on the right-hand side.

Returns The dot product.

Return type `UncertainArray`

matmul (*lhs*, *rhs*)

Matrix product of a pair of two-dimensional arrays.

For more details see `numpy.matmul`.

Parameters

- **lhs** – 2D array-like object.
- **rhs** – 2D array-like object.

Returns The matrix product.

Return type `UncertainArray`

solve (*a*, *b*)

Return x, the solution of $a \cdot x = b$

Parameters

- **a** – 2D *UncertainArray*
- **b** – *UncertainArray*

Return type *UncertainArray*

Example:

```
>>> a = la.uarray([[-2,3],[-4,1]])
>>> b = la.uarray([4,-2])
>>> la.solve(a,b)
uarray([1.0, 2.0])
```

inv (*a*)

Return the (multiplicative) matrix inverse

Example:

```
>>> x = la.uarray( [[2,1],[3,4]])
>>> x_inv =la.inv(x)
>>> la.matmul(x,x_inv)
uarray([[1.0, 0.0],
        [4.440892098500626e-16, 1.0]])
```

det (*a*)

Return the matrix determinant

Example:

```
>>> x = la.uarray( range(4) )
>>> x.shape = 2,2
>>> print(x)
[[0 1]
[2 3]]
```

(continues on next page)

(continued from previous page)

```
>>> la.det(x)
-2.0
```

identity(*n*)

Return an identity array with `n` dimensions

Example:

```
>>> la.identity(3)
uarray([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

zeros(*shape*)

Return an array of shape `shape` containing `0` elements

Example:

```
>>> la.zeros( (2,3) )
uarray([[0, 0, 0],
        [0, 0, 0]])
```

ones(*shape*)

Return an array of shape `shape` containing `1` elements

Example:

```
>>> la.ones( (2,3) )
uarray([[1, 1, 1], [1, 1, 1]])
```

empty(*shape*)

Return an array of shape `shape` containing `None` elements

Example:

```
>>> la.empty( (2,3) )
uarray([[None, None, None],
        [None, None, None]])
```

full(*shape, fill_value*)

Return an array of shape `shape` containing `fill_value` elements

Example:

```
>>> la.full( (1,3),ureal(2,1) )
uarray([[ureal(2.0,1.0,inf), ureal(2.0,1.0,inf),
         ureal(2.0,1.0,inf)]])
```

transpose(*a, axes=None*)

Array transpose

For more details see `numpy.transpose()`.

Parameters **a** – The array-like object

Returns The transpose

Return type `UncertainArray`

UncertainArray

CHAPTER 3

Examples

3.1 Examples

3.1.1 Examples using UncertainArray

Example 1. Creating an UncertainArray

The following example illustrates how to create an `UncertainArray` and how to use **GTC** functions for calculation.

Import the necessary **GTC** functions and modules

```
>>> from GTC import ureal, cos, type_a
```

Next, define the uncertain arrays

```
>>> voltages = la.uarray([ureal(4.937, 0.012), ureal(5.013, 0.008), ureal(4.986, 0.
↪014)])
>>> currents = la.uarray([ureal(0.023, 0.003), ureal(0.019, 0.006), ureal(0.020, 0.
↪004)])
>>> phases = la.uarray([ureal(1.0442, 2e-4), ureal(1.0438, 5e-4), ureal(1.0441, 3e-
↪4)])
```

We can use the *cos()* function to calculate the AC resistances

```
>>> resistances = (voltages / currents) * cos(phases)
>>> resistances
uarray([ureal(107.88283143147648,14.07416562378944,inf),
        ureal(132.69660967977737,41.90488273081293,inf),
        ureal(125.3181626494936,25.06618583901181,inf)])
```

Now, to calculate the average AC resistance we could use *type_a.mean()*, which evaluates the mean of the uncertain number values

```
>>> type_a.mean(resistances)
121.96586792024915
```

However, that is a real number, not an uncertain number. We have discarded all information about the uncertainty of each resistance!

A better calculation in this case uses *function.mean()*, which will propagate uncertainties

```
>>> fn.mean(resistances)
ureal(121.96586792024915,16.939155846751817,inf)
```

This obtains an uncertain number with a standard uncertainty of 16.939155846751817 that is the combined uncertainty of the mean of AC resistance values. We could also calculate this as

```
>>> math.sqrt(resistances[0].u**2 + resistances[1].u**2 + resistances[2].u**2)/3.0
16.939155846751817
```

Note: A Type-A evaluation of the standard uncertainty of the mean of the three resistance values is a different calculation

```
>>> type_a.standard_uncertainty(resistances)
7.356613978879885
```

The standard uncertainty evaluated here by *type_a.standard_uncertainty()* is a sample statistic calculated from the values alone. On the other hand, the standard uncertainty obtained by *function.mean()* is evaluated by propagating the input uncertainties through the calculation of the mean value. There is no reason to expect these two different calculations to yield the same result.

Example 2. Creating a Structured UncertainArray

One can also make use of the structured arrays feature of numpy to access columns in the array by *name* instead of by *index*.

Note: numpy arrays use a zero-based indexing scheme, so the first column corresponds to index 0

Suppose that we have the following `list` of data

```
>>> data = [[ureal(1, 1), ureal(2, 2), ureal(3, 3)],
...         [ureal(4, 4), ureal(5, 5), ureal(6, 6)],
...         [ureal(7, 7), ureal(8, 8), ureal(9, 9)]]
```

We can create an `UncertainArray` from this `list`

```
>>> ua = la.uarray(data)
```

When `ua` is created it is a *view* into `data` (i.e., no elements in `data` are copied)

```
>>> ua[0,0] is data[0][0]
True
```

However, if an element in `ua` is redefined to point to a new object then the corresponding element is `data` does not change

```
>>> ua[0,0] = ureal(99, 99)
>>> ua[0,0]
ureal(99.0,99.0,inf)
>>> data[0][0]
ureal(1.0,1.0,inf)
>>> ua[1,1] is data[1][1]
True
```

If we wanted to access the data in column 1 we would use the following

```
>>> ua[:,1]
uarray([ureal(2.0,2.0,inf), ureal(5.0,5.0,inf),
        ureal(8.0,8.0,inf)])
```

Alternatively, we can assign a *name* to each column so that we can access columns by *name* rather than by an *index* number *(note that we must cast each row in data to be a* `tuple` *data type)*

```
>>> ua = la.uarray([tuple(row) for row in data], names=['a', 'b', 'c'])
```

Since we chose column 1 to have the name `'b'` we can now access column 1 by its *name*

```
>>> ua['b']
uarray([ureal(2.0,2.0,inf), ureal(5.0,5.0,inf),
                ureal(8.0,8.0,inf)])
```

and then perform a calculation by using the *names* that were chosen

```
>>> ua['a'] * ua['b'] + ua['c']
uarray([ureal(5.0,4.123105625617661,inf),
        ureal(26.0,28.91366458960192,inf),
        ureal(65.0,79.7057087039567,inf)])
```

Example 3. Calibrating a Photodiode

Suppose that we have the task of calibrating the spectral response of a photodiode. We perform the following steps to acquire the data and then perform the calculation to determine the spectral response of the photodiode (PD) relative to a calibrated reference detector (REF). The experimental procedure is as follows:

1) Select a wavelength from the light source.
2) Move REF to be in the beam path of the light source.
3) Block the light and measure the background signal of REF.
4) Unblock the light and measure the signal of REF.
5) Move PD to be in the beam path of the light source.
6) Block the light and measure the background signal of PD.
7) Unblock the light and measure the signal of PD.
8) Repeat step (1).

10 readings were acquired in steps 3, 4, 6 and 7 and they were used determine the average and standard deviation for each measurement. The standard deviation is shown in brackets in the table below. The uncertainty of the wavelength is negligible.

Wavelength [nm]	**PD Signal** [Volts]	**PD Background** [Volts]	**REF Signal** [Volts]	**REF Background** [Volts]
400	1.273(4)	0.0004(3)	3.721(2)	0.00002(2)
500	2.741(7)	0.0006(2)	5.825(4)	0.00004(3)
600	2.916(3)	0.0002(1)	6.015(3)	0.00003(1)
700	1.741(5)	0.0003(4)	4.813(4)	0.00005(4)
800	0.442(9)	0.0004(3)	1.421(2)	0.00003(1)

We can create a `list` from the information in the table. It is okay to mix built-in data types (e.g., `int`, `float` or `complex`) with uncertain numbers. The degrees of freedom = 10 - 1 = 9.

```
>>> data = [
...     (400., ureal(1.273, 4e-3, 9), ureal(4e-4, 3e-4, 9), ureal(3.721, 2e-3, 9),␣
↪ureal(2e-5, 2e-5, 9)),
...     (500., ureal(2.741, 7e-3, 9), ureal(6e-4, 2e-4, 9), ureal(5.825, 4e-3, 9),␣
↪ureal(4e-5, 3e-5, 9)),
...     (600., ureal(2.916, 3e-3, 9), ureal(2e-4, 1e-4, 9), ureal(6.015, 3e-3, 9),␣
↪ureal(3e-5, 1e-5, 9)),
...     (700., ureal(1.741, 5e-3, 9), ureal(3e-4, 4e-4, 9), ureal(4.813, 4e-3, 9),␣
↪ureal(5e-5, 4e-5, 9)),
...     (800., ureal(0.442, 9e-3, 9), ureal(4e-4, 3e-4, 9), ureal(1.421, 2e-3, 9),␣
↪ureal(3e-5, 1e-5, 9))
... ]
```

Next, we create a *named* `UncertainArray` from `data` and calculate the relative spectral response by using the *names* that were specified

```
>>> ua = la.uarray(data, names=['nm', 'pd-sig', 'pd-bg', 'ref-sig', 'ref-bg'])
>>> res = (ua['pd-sig'] - ua['pd-bg']) / (ua['ref-sig'] - ua['ref-bg'])
>>> res
uarray([ureal(0.342006675660713,0.0010935674325269068,9.630065079733788),
        ureal(0.4704581662363347,0.0012448685947602906,10.30987538377716),
        ureal(0.4847571974590064,0.0005545173836499742,13.031921586772652),
        ureal(0.36167007760313324,0.0010846673083513545,10.620461706054874),
        ureal(0.31077362646642787,0.006352297390618683,9.105944114389143)])
```

Since `ua` and `res` are numpy arrays we can use numpy syntax to filter information. To select the data where the PD signal is > 2 volts, we can use

```
>>> gt2 = ua[ ua['pd-sig'] > 2 ]
>>> gt2
uarray([(500., ureal(2.741,0.007,9.0), ureal(0.0006,0.0002,9.0), ureal(5.825,0.004,
↪9.0), ureal(4e-05,3e-05,9.0)),
        (600., ureal(2.916,0.003,9.0), ureal(0.0002,0.0001,9.0), ureal(6.015,0.003,
↪9.0), ureal(3e-05,1e-05,9.0))],
        dtype=[('nm', '<f8'), ('pd-sig', 'O'), ('pd-bg', 'O'), ('ref-sig', 'O'), (
↪'ref-bg', 'O')])
```

We can also use the *name* feature on `gt2` to then get the REF signal for the filtered data

```
>>> gt2['ref-sig']
uarray([ureal(5.825,0.004,9.0), ureal(6.015,0.003,9.0)])
```

To select the relative spectral response where the wavelengths are < 700 nm

```
>>> res[ ua['nm'] < 700 ]
uarray([ureal(0.342006675660713,0.0010935674325269068,9.630065079733788),
        ureal(0.4704581662363347,0.0012448685947602906,10.30987538377716),
        ureal(0.4847571974590064,0.0005545173836499742,13.031921586772652)])
```

This is a very simplified analysis. In practise one should use a *Measurement Model*.

Example 4. N-Dimensional UncertainArrays

The multi-dimensional aspect of numpy arrays is also supported.

Suppose that we want to multiply two matrices that are composed of uncertain numbers

$$C = AB$$

The A and B matrices are defined as

```
>>> A = la.uarray([[ureal(3.6, 0.1), ureal(1.3, 0.2), ureal(-2.5, 0.4)],
...                [ureal(-0.2, 0.5), ureal(3.1, 0.05), ureal(4.4, 0.1)],
...                [ureal(8.3, 1.5), ureal(4.2, 0.6), ureal(3.3, 0.9)]])
>>> B = la.uarray([ureal(1.8, 0.3), ureal(-3.5, 0.9), ureal(0.8, 0.03)])
```

Using the `@` operator for matrix multiplication, which was introduced in Python 3.5 (**PEP 465**), we can determine C

```
>>> C = A @ B
>>> C
uarray([ureal(-0.0699999999999994,1.7792484368406793,inf),
        ureal(-7.689999999999999,2.9414535522424963,inf),
        ureal(2.8800000000000003,5.719851484085929,inf)])
```

Alternatively, we can use `matmul()` from the `linear_algebra` module

```
>>> C = la.matmul(A, B)
>>> C
uarray([ureal(-0.0699999999999994,1.7792484368406793,inf),
        ureal(-7.689999999999999,2.9414535522424963,inf),
        ureal(2.8800000000000003,5.719851484085929,inf)])
```

CHAPTER 4

Release Notes

4.1 License

4.2 Developers

- Blair Hall
- Joseph Borbely

4.3 Release Notes

4.3.1 Version 1.1.0 (2019.05.30)

- Mathematical functions in the *core* module (`sin`, `sqrt`, etc) can be applied to Python numbers as well as uncertain numbers (previously these functions raised an exception when applied to Python numbers).
- There is a new array-like class to hold collections of uncertain numbers. *UncertainArray* is based on `numpy.ndarray`, which provides excellent support for manipulating stored data. Standard mathematical operations in the *core* module can be applied to *UncertainArray* objects.
- A function *reporting.sensitivity()* calculates partial derivatives (sensitivity coefficients).

4.3.2 Version 1.0.0 (2018.11.16)

The initial release of the Python code version of the GUM Tree Calculator.

The source code was derived from the stand-alone GUM Tree Calculator version 0.9.11, which is available from the MSL web site . The new version has made some significant changes to the data structures used, with accompanying changes to the underlying algorithms.

The application programmer interface in GTC 1.0.0 remains very close to that provided in GTC 0.9.11, although not all functions in GTC 0.9.11 are available yet. It is our intention to provide the remainder in forthcoming releases.

The most significant change has been to the method of storing uncertain numbers. The `archive` module in GTC 0.9.11 was replaced in GTC 1.0.0 by the `persistence` module. So, archives created using GTC 0.9.11 are not interchangeable with GTC 1.0.0.

4.4 Indices and tables

- genindex
- modindex

Python Module Index

c

f

l

p

r

t

Index

Symbols

A

B

C

D

E

F

G

I

K