GUM Tree Calculator Documentation

Release 1.3.8

Measurement Standards Laboratory of New Zealand

Apr 01, 2022

CONTENTS

CHAPTER

ONE

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))
...
```

(continues on next page)

(continued from previous page)

```
E_phi: 0.843
E_b: 0.023
```

CHAPTER

TWO

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 *`).

acos(x)

Uncertain number arc-cosine function

Note: There is a singularity in the uncertainty if the value of `x` is unity. For example,

```
>>> acos(ureal(1, 1))
Traceback (most recent call last):
...
ZeroDivisionError: float division by zero
```

More generally, uncertainty in the inverse cosine function becomes large when x is close to unity. This may be an indication that the measurement model is ill-formed or that a GUM calculation of uncertainty is not appropriate.

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.

---

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.

asin(x)

Uncertain number arcsine function

Note: There is a singularity in the uncertainty if the value of **x** is unity. For example,

```
>>> asin(ureal(1, 1))
Traceback (most recent call last):
...
ZeroDivisionError: float division by zero
```

More generally, uncertainty in the inverse sine function becomes large when x is close to unity. This may be an indication that the measurement model is ill-formed or that a GUM calculation of uncertainty is not appropriate.

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.

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.

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∞, continuous from the right. The other extends from $-$j along the imaginary axis to $-$j∞, 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
```

(continues on next page)

(continued from previous page)

```
ureal(0.5235987755982989,1.0,inf)
>>> math.degrees( theta.x )
30.000000000000004
```

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 $-\infty$, continuous from above.

---

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
```

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')
```

cos(*x*)

Uncertain number cosine function

cosh(*x*)

Uncertain number hyperbolic cosine function

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
>>> ur.df
3.0

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

exp(*x*)

Uncertain number exponential function

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.

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.

label(*x*)

Return the label

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.

---

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)`.

---

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)
>>> 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)
```

(continues on next page)

(continued from previous page)

```
>>> z.r
-0.591484610818998...
```

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)
```

phase(*z*)

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

Returns the phase in radians

Return type *UncertainReal*

pow(*x*, *y*)

Uncertain number power function

Raises x to the power of y

result(*un*, *label=None*)

Create an uncertain number to represent an intermediate result

Parameters

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

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

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

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 uncertain numbers to be stored in an archive.

Note: This function will usually be applied to a temporary object.

Note: Applying this function to an elementary uncertain number does not create an intermediate intermediate result. If a label is already assigned to the elementary uncertain number, it will not be changed and a warning will be raised, otherwise *result()* will assign a label.

Example:

```
>>> I = ureal(1.3E-3,0.01E-3)
>>> R = ureal(995,7)
>>> V = result( I*R )
>>> P = V**2/R
>>> component(P,V)
3.5057845056420 68e-05
```

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)
```

sin(x)

Uncertain number sine function

sinh(x)

Uncertain number hyperbolic sine function

sqrt(x)

Uncertain number square root function

Note: There is a singularity in the uncertainty if the value of `x` is zero. For example,

```
>>> sqrt(ureal(0,1))
Traceback (most recent call last):
...
ZeroDivisionError: float division by zero
```

More generally, uncertainty in $\sqrt{x}$ becomes large when x is close to zero. This may be an indication that the measurement model is ill-formed or that a GUM calculation of uncertainty is not appropriate.

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

---

tan(x)

Uncertain number tangent function

tanh(x)

Uncertain number hyperbolic tangent function

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)
```

uid(*x*)

Return the GTC unique identifier for the uncertain number or `None`

New in version 1.3.7.

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)
```

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')
```

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
```

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)
```

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*

property df

Return the degrees of freedom

Return type float

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

Example:

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

property imag

Returns the imaginary component

Return type *UncertainReal*

property label

The uncertain-number label

Return type str

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

Example:

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

property real

Return the real component

Return type *UncertainReal*

property u

Return the standard uncertainty

Return type float

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

Example:

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

property uid

Return the unique identifier for the uncertain number, or `None`.

Note that `un.uid` is equivalent to *uid(un)*

New in version 1.3.7.

property v

Return the standard variance

Return type float

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

Example:

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

property x

Return the value

Return type float

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

Example:

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

property 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
```

(continues on next page)

(continued from previous page)

```
>>> dof(uc)
3.0
```

imag

The imaginary component.

Type *UncertainReal*

property label

The uncertain-number label

Return type str

Note that `uc.label` is equivalent to *label(uc)*

Example:

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

property r

Return the correlation coefficient between real and imaginary components

Return type float

real

The real component.

Type *UncertainReal*

property u

Return standard uncertainties for the real and imaginary components

Return type *StandardUncertainty*

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)
>>> uncertainty(uc)
StandardUncertainty(real=0.5, imag=0.5)
```

property uid

Return the unique identifier for the uncertain number, or `None`.

Note that `un.uid` is equivalent to *uid(un)*

New in version 1.3.7.

property 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 *VarianceCovariance*

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)
>>> variance(uc)
VarianceCovariance(rr=0.25, ri=0.0, ir=0.0, ii=0.25)
```

property 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)
>>> value(uc)
(1+2j)
```

2.2 Evaluating type-A uncertainty

Type-A evaluation of uncertainty involves statistical analysis of data (in contrast to type-B evaluation, which uses 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.

2.2.2 Least squares regression

- *line_fit()* performs an ordinary least-squares straight line fit to a sample of data.
- *line_fit_wls()* performs a weighted least-squares straight line fit to a sample of data. The weights are assumed to be exact.
- *line_fit_rwls()* performs a weighted least-squares straight line fit to a sample of data. The weights are only assumed normalise the variability of observations.
- *line_fit_wtls()* performs a weighted total least-squares straight line fit to a sample of data.

2.2.3 Merging uncertain components

- *merge()* combines results from a type-A and type-B analysis of the same 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.

merge() is provided so that the results of type-A and type-B analyses on the same data can be combined.

2.2.4 Module contents

class LineFitOLS(*a*, *b*, *ssr*, *N*)

Class to hold the results of an ordinary least-squares regression to data.

It can also be used to apply the results of a regression analysis.

New in version 1.2.

property N

The number of points in the sample

property a_b

Return the intercept `a` and slope `b` as a tuple of uncertain numbers

property intercept

Return the intercept as an uncertain number.

property slope

Return the slope as an uncertain number.

property ssr

Sum of the squared residuals

The sum of the squared deviations between values predicted by the model and the actual data.

If weights are used during the fit, the squares of weighted deviations are summed.

x_from_y(*yseq*, *x_label=None*, *y_label=None*)

Estimate the stimulus `x` corresponding to the responses in `yseq`

Parameters

- **yseq** – a sequence of `y` observations
- **x_label** – a label for the return uncertain number *x*
- **y_label** – a label for the estimate of *y* based on `yseq`

Note: When `x_label` is defined, the uncertain number returned will be declared an intermediate result (using *result()*)

Example

```
>>> x_data = [0.1, 0.1, 0.1, 0.3, 0.3, 0.3, 0.5, 0.5, 0.5,
...                 0.7, 0.7, 0.7, 0.9, 0.9, 0.9]
>>> y_data = [0.028, 0.029, 0.029, 0.084, 0.083, 0.081, 0.135, 0.131,
...                 0.133, 0.180, 0.181, 0.183, 0.215, 0.230, 0.216]

>>> fit = type_a.line_fit(x_data,y_data)

>>> x0 = fit.x_from_y( [0.0712, 0.0716] )
>>> x0
ureal(0.2601659751037...,0.01784461112558...,13.0)
```

y_from_x(*x*, *s_label=None*, *y_label=None*)

Return an uncertain number `y` that predicts the response to `x`

Parameters

- **x** – a real number, or an uncertain real number
- **s_label** – a label for an elementary uncertain number associated with observation variability
- **y_label** – a label for the return uncertain number *y*

This is a prediction of a single future response `y` to a stimulus `x`

The variability in observations is based on residuals obtained during regression.

An uncertain real number can be used for `x`, in which case the associated uncertainty will also be propagated into `y`.

Note: When `y_label` is defined, the uncertain number returned will be declared an intermediate result (using *result()*)

class LineFitRWLS(*a*, *b*, *ssr*, *N*)

Class to hold the results of a relative weighted least-squares regression. The weights provided normalise the variability of observations.

New in version 1.2.

property N

The number of points in the sample

property a_b

Return the intercept `a` and slope `b` as a tuple of uncertain numbers

property intercept

Return the intercept as an uncertain number.

property slope

Return the slope as an uncertain number.

property ssr

Sum of the squared residuals

The sum of the squared deviations between values predicted by the model and the actual data.

If weights are used during the fit, the squares of weighted deviations are summed.

x_from_y(*yseq*, *s_y*, *x_label=None*, *y_label=None*)

Estimate the stimulus `x` corresponding to the responses in `yseq`

Parameters

- **yseq** – a sequence of further observations of `y`
- **s_y** – a scale factor for the uncertainty of the `yseq`
- **x_label** – a label for the return uncertain number *x*
- **y_label** – a label for the estimate of *y* based on `yseq`

Note: When `x_label` is defined, the uncertain number returned will be declared an intermediate result (using *result()*)

---

y_from_x(*x*, *s_y*, *s_label=None*, *y_label=None*)

Return an uncertain number `y` that predicts the response to `x`

Parameters

- **x** – a real number, or an uncertain real number
- **s_y** – a scale factor for the response uncertainty
- **s_label** – a label for an elementary uncertain number associated with observation variability
- **y_label** – a label for the return uncertain number *y*

Returns a single future response `y` predicted for a stimulus `x`.

Because there is different variability in the response to different stimuli, the scale factor `s_y` is required. It is assumed that the standard deviation in the `y` value is proportional to `s_y`.

An uncertain real number can be used for `x`, in which case the associated uncertainty is also propagated into `y`.

Note: When `y_label` is defined, the uncertain number returned will be declared an intermediate result (using *result()*)

---

class LineFitWLS(*a*, *b*, *ssr*, *N*)

Class to hold the results of a weighted least-squares regression. The weight factors provided are assumed to correspond exactly to the variability of observations.

New in version 1.2.

property N

The number of points in the sample

property a_b

Return the intercept `a` and slope `b` as a tuple of uncertain numbers

property intercept

Return the intercept as an uncertain number.

property slope

Return the slope as an uncertain number.

property ssr

Sum of the squared residuals

The sum of the squared deviations between values predicted by the model and the actual data.

If weights are used during the fit, the squares of weighted deviations are summed.

x_from_y(*y_data*, *u_y_data*, *x_label=None*, *y_label=None*)

Estimate the stimulus `x` corresponding to the responses in `y_data`

Parameters

- **y_data** – a sequence of further observations of `y`
- **u_y_data** – the standard uncertainty of the `y_data` elements
- **x_label** – a label for the return uncertain number *x*
- **y_label** – a label for the estimate of *y* based on `y_data`

The variations in `y_data` values are assumed to result from independent random effects.

Note: When `x_label` is defined, the uncertain number returned will be declared an intermediate result (using *result()*)

---

y_from_x(*x*, *s_y*, *s_label=None*, *y_label=None*)

Return an uncertain number `y` that predicts the response to `x`

Parameters

- **x** – a real number, or an uncertain real number
- **s_y** – response variability uncertainty
- **s_label** – a label for an elementary uncertain number associated with response variability
- **y_label** – a label for the return uncertain number *y*

Returns a single future response `y` predicted for a stimulus `x`.

The standard uncertainty `s_y` is used to create an additive component of uncertainty associated with variability in the `y` value.

An uncertain real number can be used for `x`, in which case the associated uncertainty is also propagated into `y`.

Note: When `y_label` is defined, the uncertain number returned will be declared an intermediate result (using *result()*)

---

class LineFitWTLS(*a*, *b*, *ssr*, *N*)

This object holds results from a TLS linear regression to data.

New in version 1.2.

property N

The number of points in the sample

property a_b

Return the intercept `a` and slope `b` as a tuple of uncertain numbers

property intercept

Return the intercept as an uncertain number.

property slope

Return the slope as an uncertain number.

property ssr

Sum of the squared residuals

The sum of the squared deviations between values predicted by the model and the actual data.

If weights are used during the fit, the squares of weighted deviations are summed.

estimate(*seq*, *label=None*)

Return an uncertain number for the mean of the data in `seq`

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.

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

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).

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=-0.3137404512459525, df=9)
```

estimate_digitized(*seq*, *delta*, *label=None*, *truncate=False*)

Return an uncertain number for the mean of digitized data in `seq`

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 evaluates the mean of the data and evaluates the uncertainty in this mean.

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(-0.005627272727272...,1.9497827808661...e-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(-0.005636363636363...,1.52120004824377...e-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(-0.0056,2.8867513459481...e-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(-0.0056,3.2914029430219...e-05,2)
```

line_fit(*x*, *y*, *label=None*)

Return a least-squares straight-line fit to the data

New in version 1.2.

Parameters

- **x** – sequence of stimulus data (independent-variable)
- **y** – sequence of response data (dependent-variable)
- **label** – suffix to label the uncertain numbers *a* and *b*

Returns an object containing regression results

Return type *LineFitOLS*

Performs an ordinary least-squares regression of `y` to `x`.

Example:

```
>>> x = [1,2,3,4,5,6,7,8,9]
>>> y = [15.6,17.5,36.6,43.8,58.2,61.6,64.2,70.4,98.8]
>>> result = type_a.line_fit(x,y)
>>> a,b = result.a_b
>>> a
ureal(4.8138888888888...,4.8862063121833...,7)
>>> b
ureal(9.4083333333333...,0.8683016476563...,7)

>>> y_p = a + b*5.5
>>> dof(y_p)
7.0
```

line_fit_rwls(*x*, *y*, *s_y*, *label=None*)

Return a relative weighted least-squares straight-line fit

New in version 1.2.

The `s_y` values are used to scale variability in the `y` data. It is assumed that the standard deviation of each `y` value is proportional to the corresponding `s_y` scale factor. The unknown common factor in the uncertainties is estimated from the residuals.

Parameters

- **x** – sequence of stimulus data (independent-variable)
- **y** – sequence of response data (dependent-variable)
- **s_y** – sequence of scale factors
- **label** – suffix to label the uncertain numbers *a* and *b*

Returns an object containing regression results

Return type *LineFitRWLS*

Example:

```
>>> x = [1,2,3,4,5,6]
>>> y = [3.014,5.225,7.004,9.061,11.201,12.762]
>>> s_y = [0.2,0.2,0.2,0.4,0.4,0.4]
>>> fit = type_a.line_fit_rwls(x,y,s_y)
>>> a, b = fit.a_b
>>>
>>> print(fit)

Relative Weighted Least-Squares Results:

  Intercept: 1.14(12)
  Slope: 1.973(41)
  Correlation: -0.87
  Sum of the squared residuals: 1.3395217958...
  Number of points: 6
```

line_fit_wls(*x*, *y*, *u_y*, *label=None*)

Return a weighted least-squares straight-line fit

New in version 1.2.

Parameters

- **x** – sequence of stimulus data (independent-variable)
- **y** – sequence of response data (dependent-variable)
- **u_y** – sequence of uncertainties in the response data
- **label** – suffix to label the uncertain numbers *a* and *b*

Returns an object containing regression results

Return type *LineFitWLS*

Example:

```
>>> x = [1,2,3,4,5,6]
>>> y = [3.2, 4.3, 7.6, 8.6, 11.7, 12.8]
>>> u_y = [0.5,0.5,0.5,1.0,1.0,1.0]

>>> fit = type_a.line_fit_wls(x,y,u_y)
>>> fit.a_b
 InterceptSlope(a=ureal(0.8852320675105...,0.5297081435088...,inf),
 b=ureal(2.056962025316...,0.177892016741...,inf))
```

line_fit_wtls(*x*, *y*, *u_x*, *u_y*, *a0_b0=None*, *r_xy=None*, *label=None*)

Return a total least-squares straight-line fit

New in version 1.2.

Parameters

- **x** – sequence of independent-variable data
- **y** – sequence of dependent-variable data
- **u_x** – sequence of uncertainties in x
- **u_y** – sequence of uncertainties in y
- **a0_b0** – a pair of initial estimates for the intercept and slope
- **r_xy** – correlation between x-y pairs
- **label** – suffix labeling the uncertain numbers *a* and *b*

Returns an object containing the fitting results

Return type *LineFitWTLS*

The optional argument a_b can be used to provide a pair of initial estimates for the intercept and slope.

Based on paper by M Krystek and M Anton, *Meas. Sci. Technol.* **22** (2011) 035101 (9pp)

Example:

```
# Pearson-York test data see, e.g.,
# Lybanon, M. in Am. J. Phys 52 (1) 1984
>>> x=[0.0,0.9,1.8,2.6,3.3,4.4,5.2,6.1,6.5,7.4]
>>> wx=[1000.0,1000.0,500.0,800.0,200.0,80.0,60.0,20.0,1.8,1.0]

>>> y=[5.9,5.4,4.4,4.6,3.5,3.7,2.8,2.8,2.4,1.5]
>>> wy=[1.0,1.8,4.0,8.0,20.0,20.0,70.0,70.0,100.0,500.0]

# standard uncertainties required for weighting
>>> ux=[1./math.sqrt(wx_i) for wx_i in wx ]
>>> uy=[1./math.sqrt(wy_i) for wy_i in wy ]

>>> result = ta.line_fit_wtls(x,y,ux,uy)
>>> intercept, slope = result.a_b
>>> intercept
ureal(5.47991018...,0.29193349...,8)
>>> slope
ureal(-0.48053339...,0.057616740...,8)
```

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
```

Note: When `seq` is an empty `ndarray` or a `ndarray` containing any `NaN` elements `NaN` is returned.

In other cases, a `ZeroDivisionError` is raised when there are no elements in `seq`.

---

merge(*a*, *b*, *TOL=1e-13*)

Combine the uncertainty components of `a` and `b`

Parameters

- **a** – an uncertain real or complex number
- **b** – an uncertain real or complex number
- **TOL** – float

Returns an uncertain number with the value of `a` and the uncertainty components of `a` and `b` combined

The absolute difference between the values of `a` and `b` must be less than `TOL` and the components of uncertainty associated with `a` and `b` must be distinct, otherwise a `RuntimeError` will be raised.

Use this function to combine results from type-A and type-B uncertainty analyses performed on a common sequence of data.

Note: Some judgement will be required as to when it is appropriate to merge uncertainty components.

There is a risk of 'double-counting' uncertainty if type-B components are contributing to the variability observed in the data, and therefore assessed in a type-A analysis.

Changed in version 1.3.3: Added the *TOL* keyword argument.

Example:

```
# From Appendix H3 in the GUM

# Thermometer readings (degrees C)
t = (21.521,22.012,22.512,23.003,23.507,
    23.999,24.513,25.002,25.503,26.010,26.511)

# Observed differences with calibration standard (degrees C)
b = (-0.171,-0.169,-0.166,-0.159,-0.164,
    -0.165,-0.156,-0.157,-0.159,-0.161,-0.160)

# Arbitrary offset temperature (degrees C)
t_0 = 20.0

# Calculate the temperature relative to t_0
t_rel = [ t_k - t_0 for t_k in t ]

# A common systematic error in all differences
e_sys = ureal(0,0.01)

b_type_b = [ b_k + e_sys for b_k in b ]

# Type-A least-squares regression
```

(continues on next page)

(continued from previous page)

```
y_1_a, y_2_a = type_a.line_fit(t_rel,b_type_b).a_b

# Type-B least-squares regression
y_1_b, y_2_b = type_b.line_fit(t_rel,b_type_b)

# `y_1` and `y_2` have uncertainty components
# related to the type-A analysis as well as the
# type-B systematic error
y_1 = type_a.merge(y_1_a,y_1_b)
y_2 = type_a.merge(y_2_a,y_2_b)
```

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

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 data that is associated with a particular quantity. An uncertain number for that quantity will be created from sample statistics. The covariance between the different quantities will also be evaluated.

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

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...
```

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 contains a sample of data associated with a particular quantity. An uncertain number will be created for the quantity from sample statistics. The covariance between the different quantities will also be evaluated.

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

Example:

```
# From Appendix H2 in the GUM

>>> 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(4.999000...,0.0032093613071761...,4, label='V')
>>> i
ureal(0.019661,9.471008394041335...e-06,4, label='I')
>>> p
ureal(1.044460...,0.0007520638270785...,4, label='phi')

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

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 associated with 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 Least-squares regression

line_fit() implements an ordinary least-squares straight-line regression calculation that accepts uncertain real numbers for the independent and dependent variables.

line_fit_wls() implements a weighted least-squares straight-line regression calculation. It accepts uncertain real numbers for the independent and dependent variables. It is also possible to specify weights for the regression.

line_fit_wtls() implements a total least-squares algorithm for a straight-line fitting that can perform a weighted least-squares regression when both y and x data are uncertain real numbers, it also handles correlation between (x,y) data pairs.

2.3.2 Probability distributions: real-valued problems

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

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

2.3.3 Probability distributions: complex-valued problems

Functions that convert information about two-dimensional distributions into standard uncertainties:

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

2.3.4 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*

2.3.5 Module contents

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`

An uncertain number is returned if `seq` contains uncertain numbers.

Example

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

Note: When `seq` is an empty `ndarray` or a `ndarray` containing any `NaN` elements `NaN` is returned.

In other cases, a `ZeroDivisionError` is raised when there are no elements in `seq`.

---

line_fit(*x*, *y*)

Least-squares fit intercept and slope

New in version 1.2.

Parameters

- **x** – sequence of independent variable data
- **y** – sequence of dependent variable data

Return type a *LineFitOLS*

y must be a sequence of uncertain real numbers.

Performs an ordinary least-squares regression.

Note: Uncertainty in the parameter estimates is found by propagation *through* the regression formulae. This does **not** take residuals into account.

The function *type_a.line_fit()* performs a regression analysis that evaluates uncertainty in the parameter estimates using the residuals.

If appropriate, the results from both type-A and type-B analyses can be merged (see *type_a.merge()*).

Example:

```
>>> a0 =10
>>> b0 = -3
>>> u0 = .2

>>> x = [ float(x_i) for x_i in xrange(10) ]
>>> y = [ ureal(b0*x_i + a0,u0) for x_i in x ]

>>> a,b = tb.line_fit(x,y).a_b
>>> a
ureal(10.0,0.1175507627290...,inf)
>>> b
ureal(-3.0,0.02201927530252...,inf)
```

line_fit_wls(*x*, *y*, *u_y=None*)

Weighted least-squares linear regression

New in version 1.2.

Parameters

- **x** – sequence of independent variable data
- **y** – sequence of dependent variable data
- **u_y** – sequence of uncertainties in y

Return type *LineFitWLS*

y must be a sequence of uncertain real numbers.

Performs a weighted least-squares regression.

Weights are calculated from the uncertainty of the y elements unless the sequence u_y is provided.

Note: The uncertainty in the parameter estimates is found by propagation of uncertainty *through* the regression formulae. This does **not** take account of the residuals.

The function *type_a.line_fit_wls()* can be used to carry out a regression analysis that obtains uncertainty in the parameter estimates due to the residuals.

If necessary, the results of both type-A and type-B analyses can be merged (see *type_a.merge()*).

Example:

```
>>> x = [1,2,3,4,5,6]
>>> y = [3.2, 4.3, 7.6, 8.6, 11.7, 12.8]
>>> u_y = [0.5,0.5,0.5,1.0,1.0,1.0]
>>> y = [ ureal(y_i,u_y_i) for y_i, u_y_i in zip(y,u_y) ]

>>> fit = type_b.line_fit_wls(x,y)
>>> a, b = fit.a_b
>>> a
ureal(0.8852320675105...,0.5297081435088...,inf)
>>> b
ureal(2.0569620253164...,0.1778920167412...,inf)
```

line_fit_wtls(*x*, *y*, *u_x=None*, *u_y=None*, *a_b=None*, *r_xy=None*)

Perform straight-line regression with uncertainty in `x` and `y`

New in version 1.2.

Parameters

- **x** – list of uncertain real numbers for the independent variable
- **y** – list of uncertain real numbers for the dependent variable
- **u_x** – a sequence of uncertainties for the `x` data
- **u_y** – a sequence of uncertainties for the `y` data
- **a_b** – a pair of initial estimates for the intercept and slope
- **r_xy** – correlation between x-y pairs [default: 0]

Returns a *LineFitWTLS* object

The elements of `x` and `y` must be uncertain numbers with non-zero uncertainties. If specified, the optional arguments `u_x` and `u_y` will be used uncertainties to weight the data for the regression, otherwise the uncertainties of the uncertain numbers in the sequences are used.

The optional argument `a_b` can be used to provide a pair of initial estimates for the intercept and slope. Otherwise, initial estimates will be obtained by calling *line_fit_wls*.

Implements a Weighted Total Least Squares algorithm that allows for correlation between x-y pairs. See reference:

M Krystek and M Anton, *Meas. Sci. Technol.* **22** (2011) 035101 (9pp)

Example:

```
# Pearson-York test data
# see, e.g., Lybanon, M. in Am. J. Phys 52 (1), January 1984
>>> xin=[0.0,0.9,1.8,2.6,3.3,4.4,5.2,6.1,6.5,7.4]
>>> wx=[1000.0,1000.0,500.0,800.0,200.0,80.0,60.0,20.0,1.8,1.0]
>>> yin=[5.9,5.4,4.4,4.6,3.5,3.7,2.8,2.8,2.4,1.5]
>>> wy=[1.0,1.8,4.0,8.0,20.0,20.0,70.0,70.0,100.0,500.0]

# Convert weights to standard uncertainties
>>> uxin=[1./math.sqrt(wx_i) for wx_i in wx ]
>>> uyin=[1./math.sqrt(wy_i) for wy_i in wy ]

# Define uncertain numbers
>>> x = [ ureal(xin_i,uxin_i) for xin_i,uxin_i in zip(xin,uxin) ]
```

(continues on next page)

(continued from previous page)

```
>>> y = [ ureal(yin_i,uyin_i) for yin_i,uyin_i in zip(yin,uyin) ]

# TLS returns uncertain numbers
>>> a,b = type_b.line_fit_wtls(x,y).a_b
>>> a
ureal(5.47991018...,0.29193349...,inf)
>>> b
ureal(-0.48053339...,0.057616740...,inf)
```

class LineFitOLS(*a*, *b*, *ssr*, *N*)

Class to hold results from an ordinary linear regression to data.

New in version 1.2.

property N

The number of points in the sample

property a_b

Return the intercept `a` and slope `b` as a tuple of uncertain numbers

property intercept

Return the intercept as an uncertain number.

property slope

Return the slope as an uncertain number.

property ssr

Sum of the squared residuals

The sum of the squared deviations between values predicted by the model and the actual data.

If weights are used during the fit, the squares of weighted deviations are summed.

x_from_y(*yseq*, *x_label=None*)

Estimate the stimulus `x` corresponding to the responses in `yseq`

Parameters

- **yseq** – a sequence of further observations of `y`
- **x_label** – a label for the return uncertain number *x*

The items in `yseq` must be uncertain real numbers.

Note: When `x_label` is defined, the uncertain number returned will be declared an intermediate result (using *result()*)

---

y_from_x(*x*, *y_label=None*)

Return an uncertain number `y` that predicts the response to `x`

Parameters

- **x** – an uncertain real number
- **y_label** – a label for the return uncertain number *y*

This is a prediction of a single future response `y` to a stimulus `x`

Note: When `y_label` is defined, the uncertain number returned will be declared an intermediate result (using *result()*)

---

class LineFitWLS(*a*, *b*, *ssr*, *N*)

This object holds results from a weighted LS linear regression to data.

New in version 1.2.

property N

The number of points in the sample

property a_b

Return the intercept `a` and slope `b` as a tuple of uncertain numbers

property intercept

Return the intercept as an uncertain number.

property slope

Return the slope as an uncertain number.

property ssr

Sum of the squared residuals

The sum of the squared deviations between values predicted by the model and the actual data.

If weights are used during the fit, the squares of weighted deviations are summed.

x_from_y(*yseq*, *x_label=None*)

Estimate the stimulus `x` corresponding to the responses in `yseq`

Parameters

- **yseq** – a sequence of further observations of `y`
- **x_label** – a label for the return uncertain number *x*

The items in `yseq` must be uncertain real numbers.

Note: When `x_label` is defined, the uncertain number returned will be declared an intermediate result (using *result()*)

---

y_from_x(*x*, *y_label=None*)

Return an uncertain number `y` that predicts the response to `x`

Parameters

- **x** – an uncertain real number
- **y_label** – a label for the return uncertain number *y*

This is a prediction of a single future response `y` to a stimulus `x`

Note: When `y_label` is defined, the uncertain number returned will be declared an intermediate result (using *result()*)

---

class LineFitWTLS(*a*, *b*, *ssr*, *N*)

This object holds results from a TLS linear regression to data.

New in version 1.2.

property N

The number of points in the sample

property a_b

Return the intercept `a` and slope `b` as a tuple of uncertain numbers

property intercept

Return the intercept as an uncertain number.

property slope

Return the slope as an uncertain number.

property ssr

Sum of the squared residuals

The sum of the squared deviations between values predicted by the model and the actual data.

If weights are used during the fit, the squares of weighted deviations are summed.

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)
```

distribution

Maps `distribution[name]` → *function* for standard uncertainty

Example:

```
>>> type_b.distribution['arcsine'](1.5)
1.06066017177982...
```

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.

The function *implicit()* will evaluate the solution to $fn(x) = 0$

2.4.2 Module contents

complex_to_seq(*z*)

Transform a complex number into a 4-element sequence

Parameters **z** – a number

This function produces a sequence of the form `[x, -y, y, x]` for a complex number `z = x + yj`.

See also *seq_to_complex()*

Example:

```
>>> 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.]]
```

Note: A matrix of the form `[[x,-y],[y,x]]` can be used to represent `z` in matrix computations.

Example:

```
>>> z = 1 + 2j
>>> m1 = numpy.array( function.complex_to_seq(z) )
>>> m1.shape = (2,2)
>>> m2 = numpy.array( function.complex_to_seq( z.conjugate() ) )
>>> m2.shape = (2,2)
>>> print( numpy.matmul(m1,m2) )
[[5. 0.]
 [0.  5.]]
```

---

implicit(*fn*, *x_min*, *x_max*, *epsilon=1e-13*)

Return the solution to $fn(x) = 0$

Parameters

- **fn** – a user-defined function of one argument
- **x_min** (*float*) – lower limit of search range
- **x_max** (*float*) – upper limit of search range
- **epsilon** (*float*) – tolerance for algorithm convergence

`x_min` and `x_max` delimit a range containing a single root (ie, the function must cross the x-axis just once inside the range).

Note:

- A `RuntimeError` is raised if the search algorithm fails to converge.
- An `AssertionError` is raised if preconditions are not satisfied.

Example:

```
>>> near_unity = ureal(1,0.05)
>>> fn = lambda x: x**2 - near_unity
>>> function.implicit(fn,0,2)
ureal(1.0,0.025...,inf)
```

New in version 1.3.4.

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`

An uncertain number is returned if `seq` contains uncertain numbers.

Example

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

Note: When `seq` is an empty `ndarray` or a `ndarray` containing any `NaN` elements `NaN` is returned.

In other cases, a `ZeroDivisionError` is raised when there are no elements in `seq`.

---

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

The legitimate form of elements in `seq` is `[x, -y, y, x]`, where `x` is the real component and `y` is the imaginary component of a complex number.

See also *complex_to_seq()*

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)
```

2.5 Reporting

2.5.1 Reporting

This module provides functions to facilitate reporting information about uncertainty calculations.

The abbreviation `rp` is defined as an alias for *reporting*, to resolve the names of objects defined in this module.

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.

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 often called a sensitivity coefficient.

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.

Module contents

budget(*y*, ***kwargs*)

Return a sequence of *Influence* objects

Parameters

- **y** – *UncertainReal* or *UncertainComplex*: an uncertain number
- ****kwargs** – Keyword arguments:
  - influences: a sequence of uncertain numbers
  - key (`str`): a sorting key (`'u'` or `'label'`)
  - reverse (`bool`): the sorting order (forward or reverse)
  - trim (`float`): to control the smallest reported magnitudes
  - max_number (`int`): to return no more than *max_number* components
  - intermediate (`bool`): to report all intermediate components

Returns A sequence of *Influence* namedtuples.

Each *Influence* has three attributes: `label`, `u`, `uid`.

- `label` is the label assigned to the uncertain number.
- `u` is the value of the component of uncertainty (see *component()*);
- `uid` is the unique identifier for the uncertain number.

The keyword argument `influences` can be used to report specific influences.

The keyword argument `key` sets the sequence ordering to use the component of uncertainty or the label, respectively, `u` or `label`.

The keyword argument `reverse` controls the sense of ordering.

The keyword argument `trim` can be used to set the minimum relative magnitude of components returned. Components of uncertainty greater than `trim` times the largest component returned will be reported. Set `trim=0` for a complete list.

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

The keyword argument `intermediate` will cause all components of uncertainty with respect to all intermediate results to be reported. When `intermediate` is `True`, `influences` cannot be specified.

Examples:

```
>>> 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 i in reporting.budget(y):
...     print("{0}: {1:G}".format(i.label,i.u))
...
x1: 0.333333
x2: 0.166667
x3: 0.0111111

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

>>> y1 = result(x1 + x2,label='y1')
>>> y2 = result(x2 + x3,label='y2')
>>> for i in reporting.budget(y1 + y2,intermediate=True):
...     print("{0}: {1:G}".format(i.label,i.u))
...
y1: 1.11803
y2: 0.509902
```

Changed in version 1.3.7: The *Influence* namedtuple has a third attribute *uid*

Changed in version 1.3.4: Added the *intermediate* keyword argument.

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
```

is_ureal(*x*)

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

Example:

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

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
```

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
```

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)
```

Note: This function evaluates the sensitivity (partial derivative) of one uncertain number with respect to another term x.

However, if the standard uncertainty of x is zero, the term is treated as being absent from the analytical model, so a sensitivity of 0 is reported.

For example

```
>>> z1 = ucomplex(1+2j,[0,1])
>>> z2 = ucomplex(-1.2-0.9j,[1,0])
>>> z = z1*z2
>>> rp.sensitivity(z,z1.real)
JacobianMatrix(rr=0.0, ri=0.0, ir=0.0, ii=0.0)
>>> rp.sensitivity(z,z1.imag)
JacobianMatrix(rr=0.9, ri=0.0, ir=-1.2, ii=0.0)
```

(continues on next page)

(continued from previous page)

```
>>> rp.sensitivity(z,z2.real)
JacobianMatrix(rr=1.0, ri=0.0, ir=2.0, ii=0.0)
>>> rp.sensitivity(z,z2.imag)
JacobianMatrix(rr=0.0, ri=0.0, ir=0.0, ii=0.0)
```

If all the partial derivatives of a measurement model are required, regardless of the associated standard uncertainties, the preferred method is to assign all standard uncertainty values to unity.

Using the same example as above

```
>>> z1 = ucomplex(1+2j,1)
>>> z2 = ucomplex(-1.2-0.9j,1)
>>> z = z1*z2
>>> rp.sensitivity(z,z1.real)
JacobianMatrix(rr=-1.2, ri=0.0, ir=-0.9, ii=0.0)
>>> rp.sensitivity(z,z1.imag)
JacobianMatrix(rr=0.9, ri=0.0, ir=-1.2, ii=0.0)
>>> rp.sensitivity(z,z2.real)
JacobianMatrix(rr=1.0, ri=0.0, ir=2.0, ii=0.0)
>>> rp.sensitivity(z,z2.imag)
JacobianMatrix(rr=-2.0, ri=0.0, ir=1.0, ii=0.0)
```

---

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
```

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)
```

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
```

2.5.2 Managing information about influences

GTC uncertainty calculations produce a final value, a standard uncertainty, degrees of freedom, and components of uncertainty.

To assist in the display of uncertainty budgets, GTC also allows elementary uncertain numbers, and intermediate uncertain numbers, to be labelled.

If additional information about influence quantities or intermediate results is needed, unique identifiers can be used to index the stored data. These identifiers remain unique in different sessions and on different computers. So, additional details may be collected and stored for later use.

A typical use-case is to retain information about the name of the distribution associated with the uncertainty in each influence quantity value. This might be expected when reporting an uncertainty budget.

Here is a re-working of an example from the GUM (see *Gauge block measurement (GUM H1)*), which shows how to handle such cases.

We define a namedtuple to hold extra data and a dictionary to hold tuples for each influence, indexed by unique identifiers.

The calculation is the same as in *Gauge block measurement (GUM H1)*. However, a label and the type of distribution associated with each influence are now stored in `manifest`, which is used when displaying the results.

```
from collections import namedtuple
from GTC import *

InfluenceData = namedtuple('InfluenceData','label, distribution')
manifest = dict()

# Lengths in nm
d0 = ureal(215,5.8,24)
d1 = ureal(0.0,3.9,5)
d2 = ureal(0.0,6.7,8)

manifest[d0.uid] = InfluenceData(label='d0',distribution='normal')
manifest[d1.uid] = InfluenceData(label='d1',distribution='normal')
manifest[d2.uid] = InfluenceData(label='d2',distribution='normal')

# Intermediate quantity 'd'
d = d0 + d1 + d2

alpha_s = ureal(11.5E-6, type_b.uniform(2E-6))
d_alpha = ureal(0.0, type_b.uniform(1E-6), 50)
d_theta = ureal(0.0, type_b.uniform(0.05), 2)

manifest[alpha_s.uid] = InfluenceData(label='alpha_s',distribution='uniform')
manifest[d_alpha.uid] = InfluenceData(label='d_alpha',distribution='uniform')
manifest[d_theta.uid] = InfluenceData(label='d_theta',distribution='uniform')

theta_bar = ureal(-0.1,0.2)
Delta = ureal(0.0, type_b.arcsine(0.5))

manifest[theta_bar.uid] = InfluenceData(label='theta_bar',distribution='normal')
manifest[Delta.uid] = InfluenceData(label='Delta',distribution='arcsine')

# Intermediate quantity 'theta'
theta = theta_bar + Delta

l_s = ureal(5.0000623E7,25,18)

manifest[l_s.uid] = InfluenceData(label='l_s',distribution='normal')

# two more intermediate steps
tmp1 = l_s * d_alpha * theta
tmp2 = l_s * alpha_s * d_theta

# Final equation for the measurement result
l = result( l_s + d - (tmp1 + tmp2), label='l')

print( "Measurement result: l={} nm".format(l) )

print("""
Components of uncertainty in l (nm)
-----------------------------------""")

format_str = "  {:>10} {!s:>10}  : {:>7.2f}"
for i in reporting.budget(l,trim=0):
    m_i = manifest[i.uid]
    print( format_str.format(m_i.label,m_i.distribution,i.u) )
```

The final reporting step takes the items generated by *budget()* and uses the `uid` attribute to index the additional information stored in `manifest`.

The output is

```
Measurement result: l= 50000838(32) nm

Components of uncertainty in l (nm)
-----------------------------------
         l_s     normal  :   25.00
     d_theta    uniform  :   16.60
          d2     normal  :    6.70
          d0     normal  :    5.80
          d1     normal  :    3.90
     d_alpha    uniform  :    2.89
     alpha_s    uniform  :    0.00
   theta_bar     normal  :    0.00
       Delta    arcsine  :    0.00
```

2.6 The persistence module

2.6.1 Functions

Functions for storing and retrieving archive files using Python pickle format are

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

Functions for storing and retrieving pickled archive strings are

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

Functions for storing and retrieving archive files using JSON format are

- *dump_json()*
- *load_json()*

Functions for storing and retrieving an archive as a JSON-formatted string are

- *dumps_json()*
- *loads_json()*

2.6.2 Module contents

class Archive

An *Archive* helps to store and retrieve uncertain numbers, so that they can be used in later calculations.

A particular *Archive* object can either be used to prepare a record of uncertain numbers for storage, or to retrieve a stored record.

add(***kwargs*)

Add entries to an archive.

Each entry is given a name that identifies it within the archive.

Example

```
>>> a = pr.Archive()
>>> x = ureal(1,1)
>>> y = ureal(2,1)
>>> z = ureal(20,1)
>>> a.add(x=x,fred=y)

# Entries can also be added using the name as a key
>>> a['z'] = z
```

Here `f` is a file stream opened in mode 'wb':

```
>>> pr.dump(f, a)
>>> f.close()
```

extract(**args*)

Extract uncertain numbers by name

Parameters **args** – names of uncertain numbers stored in the archive

If just one name is provided, a single uncertain number is returned. Otherwise a sequence of uncertain numbers is returned.

Example

Continuing the example in *add()*, but in a different Python session, `f` is now a file stream opened in 'rb' mode:

```
>>> a = pr.load(f)
>>> f.close()

>>> a.extract('fred')
ureal(2.0,1.0,inf)
>>> x, fred = a.extract('x','fred')
>>> x
ureal(1.0,1.0,inf)

# Entries can also be extracted using the name as a key
>>> a['z']
ureal(20.0,1.0,inf)
```

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

dump(*file*, *ar*)

Save an archive in a file

Parameters

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

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

dump_json(*file*, *ar*, ***kw*)

Save an archive in a file in JSON format

Parameters

- **file** – a file object opened in text mode (with 'w')
- **ar** – an *Archive* object

Keyword arguments will be passed to `json.dump()`

Only one archive can be saved in a file.

New in version 1.3.0.

dumps(*ar*, *protocol=5*)

Save an archive pickled in a string

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.

dumps_json(*ar*, ***kw*)

Convert an archive to a JSON string

Parameters **ar** – an *Archive* object

Keyword arguments will be passed to `json.dumps()`

New in version 1.3.0.

load(*file*)

Load an archive from a file

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

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

load_json(*file*, ***kw*)

Load an archive from a file

Parameters **file** – a file created by *dump_json()*

Keyword arguments will be passed to `json.load()`

New in version 1.3.0.

loads(*s*)

Return an archive object from a pickled string

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

loads_json(*s*, ***kw*)

Return an archive object by converting a JSON string

Parameters **s** – a string created by *dumps_json()*

Keyword arguments will be passed to `json.loads()`

New in version 1.3.0.

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*, *uid*)

namedtuple: label, value, and identifier of a component of uncertainty

label

str: influence quantity label

u

float: component of uncertainty

uid

tuple: unique identifier of the uncertain number

New in version 1.3.7: The attribute *uid* has been added

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`

class InterceptSlope(*a*, *b*)

namedtuple: Values for intercept `a` and slope `b`

a

UncertainReal: intercept

b

UncertainReal: slope

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)]])
```

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

det(*a*)

Return the matrix determinant

New in version 1.1.

Example:

```
>>> x = la.uarray( range(4) )
>>> x.shape = 2,2
>>> print(x)
[[0 1]
[2 3]]
>>> la.det(x)
-2.0
```

dot(*lhs*, *rhs*)

Dot product of two arrays.

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

New in version 1.1.

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*

empty(*shape*)

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

New in version 1.1.

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

New in version 1.1.

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)]])
```

identity(*n*)

Return an identity array with `n` dimensions

New in version 1.1.

Example:

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

inv(*a*)

Return the (multiplicative) matrix inverse

New in version 1.1.

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]])
```

matmul(*lhs*, *rhs*)

Matrix product of a pair of two-dimensional arrays.

For more details see `numpy.matmul`.

New in version 1.1.

Parameters

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

Returns The matrix product.

Return type *UncertainArray*

ones(*shape*)

Return an array of shape `shape` containing 1 elements

New in version 1.1.

Example:

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

solve(*a*, *b*)

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

New in version 1.1.

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])
```

transpose(*a*, *axes=None*)

Array transpose

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

New in version 1.1.

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

Returns The transpose

Return type *UncertainArray*

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*.

New in version 1.1.

Attention: Requires numpy $\geq$ 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)])
```

zeros(*shape*)

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

New in version 1.1.

Example:

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

class UncertainArray(*array*, *dtype=None*, *label=None*)

An *UncertainArray* can contain elements of type int, float, complex, *UncertainReal* or *UncertainComplex*.

Do not instantiate this class directly. Use *uarray()* instead.

Base: numpy.ndarray

New in version 1.1.

conjugate()

The result of applying the attribute `conjugate` to each element in the array.

Example:

```
>>> a = la.uarray([ucomplex(1.2-0.5j, 0.6), ucomplex(3.2+1.2j, (1.4, 0.2)),␣
↪ucomplex(1.5j, 0.9)])
>>> a.conjugate()
uarray([ucomplex((1.2+0.5j), u=[0.6,0.6], r=0.0, df=inf),
        ucomplex((3.2-1.2j), u=[1.4,0.2], r=0.0, df=inf),
        ucomplex((0-1.5j), u=[0.9,0.9], r=0.0, df=inf)])
```

Return type *UncertainArray*

dof()

The result of *dof()* for each element in the array.

Example:

```
>>> a = la.uarray([ureal(6, 2, df=3), ureal(4, 1, df=4), ureal(5, 3, df=7),␣
↪ureal(1, 1)])
>>> a.dof()
uarray([3.0, 4.0, 7.0, inf])
```

Return type *UncertainArray*

property imag

The result of applying the attribute `imag` to each element in the array.

Example:

```
>>> a = la.uarray([ucomplex(1.2-0.5j, 0.6), ucomplex(3.2+1.2j, (1.4, 0.2)),␣
↪ucomplex(1.5j, 0.9)])
>>> a.imag
uarray([ureal(-0.5,0.6,inf), ureal(1.2,0.2,inf),
        ureal(1.5,0.9,inf)])
```

Return type *UncertainArray*

property label

The label that was assigned to the array when it was created.

Example:

```
>>> current = la.uarray([ureal(0.57, 0.18), ureal(0.45, 0.12), ureal(0.68, 0.
↪19)], label='amps')
>>> current.label
'amps'
```

Return type `str`

property r

The result of applying the attribute `r` to each element in the array.

Example:

```
>>> a = la.uarray([ucomplex(1.2-0.5j, (1.2, 0.7, 0.7, 2.2)),
...                ucomplex(-0.2+1.2j, (0.9, 0.4, 0.4, 1.5))])
>>> a.r
uarray([0.43082021842766455, 0.34426518632954817])
```

Return type *UncertainArray*

property real

The result of applying the attribute `real` to each element in the array.

Example:

```
>>> a = la.uarray([ucomplex(1.2-0.5j, 0.6), ucomplex(3.2+1.2j, (1.4, 0.2)),␣
↪ucomplex(1.5j, 0.9)])
>>> a.real
uarray([ureal(1.2,0.6,inf), ureal(3.2,1.4,inf),
        ureal(0.0,0.9,inf)])
```

Return type *UncertainArray*

uncertainty()

The result of *uncertainty()* for each element in the array.

Example:

```
>>> r = la.uarray([ureal(0.57, 0.18), ureal(0.45, 0.12), ureal(0.68, 0.19)])
>>> r.uncertainty()
uarray([0.18, 0.12, 0.19])
>>> c = la.uarray([ucomplex(1.2-0.5j, 0.6), ucomplex(3.2+1.2j, (1.4, 0.2)),␣
↪ucomplex(1.5j, 0.9)])
>>> c.uncertainty()
uarray([StandardUncertainty(real=0.6, imag=0.6),
       StandardUncertainty(real=1.4, imag=0.2),
       StandardUncertainty(real=0.9, imag=0.9)])
```

Return type *UncertainArray*

value()

The result of *value()* for each element in the array.

Example:

```
>>> a = la.uarray([0.57, ureal(0.45, 0.12), ucomplex(1.1+0.68j, 0.19)])
>>> a.value()
uarray([0.57, 0.45, (1.1+0.68j)])
```

Return type *UncertainArray*

variance()

The result of *variance()* for each element in the array.

Example:

```
>>> r = la.uarray([ureal(0.57, 0.18), ureal(0.45, 0.12), ureal(0.68, 0.19)])
>>> r.variance()
uarray([0.0324, 0.0144, 0.0361])
>>> c = la.uarray([ucomplex(1.2-0.5j, 0.6), ucomplex(3.2+1.2j, (1.5, 0.5)),␣
↪ucomplex(1.5j, 0.9)])
>>> c.variance()
uarray([VarianceCovariance(rr=0.36, ri=0.0, ir=0.0, ii=0.36),
        VarianceCovariance(rr=2.25, ri=0.0, ir=0.0, ii=0.25),
        VarianceCovariance(rr=0.81, ri=0.0, ir=0.0, ii=0.81)])
```

Return type *UncertainArray*

CHAPTER

THREE

EXAMPLES

3.1 Examples

3.1.1 GUM Appendices

Gauge block measurement (GUM H1)

An example from Appendix H1 of the GUM[1].

- *Code*
- *Explanation*

Code

```
from __future__ import print_function
from GTC import *

print("""
------------------------------
Example from Appendix H1 of GUM
------------------------------
""")

# Lengths are in nm
d0 = ureal(215,5.8,24,label='d0')
d1 = ureal(0.0,3.9,5,label='d1')
d2 = ureal(0.0,6.7,8,label='d2')

# Intermediate quantity 'd'
d = d0 + d1 + d2

alpha_s = ureal(11.5E-6, type_b.uniform(2E-6),label='alpha_s')
d_alpha = ureal(0.0, type_b.uniform(1E-6), 50,label='d_alpha')
d_theta = ureal(0.0, type_b.uniform(0.05), 2,label='d_theta')

theta_bar = ureal(-0.1,0.2,label='theta_bar')
Delta = ureal(0.0, type_b.arcsine(0.5),label='Delta')
```

(continues on next page)

[1] BIPM and IEC and IFCC and ISO and IUPAC and IUPAP and OIML, *Evaluation of measurement data - Guide to the expression of uncertainty in measurement JCGM 100:2008 (GUM 1995 with minor corrections)*, (2008) http://www.bipm.org/en/publications/guides/gum

(continued from previous page)

```
# Intermediate quantity 'theta'
theta = theta_bar + Delta

l_s = ureal(5.0000623E7,25,18,label='l_s')

# two more intermediate steps
tmp1 = l_s * d_alpha * theta
tmp2 = l_s * alpha_s * d_theta

# Final equation for the measurement result
l = result( l_s + d - (tmp1 + tmp2), label='l')

print( "Measurement result for l={}".format(l) )

print("""
Components of uncertainty in l (nm)
-----------------------------------""")

for i in reporting.budget(l):
    print( "  {!s}: {:G}".format(i.label,i.u) )
```

Explanation

The measurand is the length of an end-gauge at 20 °C. The measurement equation is[2]

$$l = l_\mathrm{s} + d - l_\mathrm{s}(\delta_\alpha\theta + \alpha_\mathrm{s}\delta_\theta) \ ,$$

where

- l_s - the length of the standard
- d - the difference in length between the standard and the end-gauge
- δ_α - the difference between coefficients of thermal expansion for the standard and the end-gauge
- θ - the deviation in temperature from 20 °C
- α_s - the coefficient of thermal expansion for the standard
- δ_θ - the temperature difference between the standard and the end-gauge

The calculation proceeds in stages. First, three inputs are defined:

- the length difference measurement(`d0`) using the comparator, which is the arithmetic mean of several indications;
- an estimate of comparator random errors (`d1`) and
- an estimate of comparator systematic errors (`d2`).

These are used to define the intermediate result `d`

```
d0 = ureal(215,5.8,24,label='d0')
d1 = ureal(0.0,3.9,5,label='d1')
d2 = ureal(0.0,6.7,8,label='d2')

# Intermediate quantity 'd'
d = d0 + d1 + d2
```

[2] In fact, the GUM uses more terms to calculate the uncertainty than are defined: quantities d and θ depend on more than one influence quantity.

Then terms are introduced to account for temperature variability and thermal properties of the gauge blocks.

In particular, the quantity θ is defined in terms of two other input quantities

$$\theta = \bar{\theta} + \Delta$$

where

- $\bar{\theta}$ is the mean deviation of the test-bed temperature from $20\,^{\circ}\mathrm{C}$
- Δ is a cyclical error in the test-bed temperature

In defining these inputs, functions *type_b.uniform()* and *type_b.arcsine()* convert the widths of particular error distributions into standard uncertainties[3].

```
alpha_s = ureal( 11.5E-6, type_b.uniform(2E-6), label='alpha_s' )
d_alpha = ureal(0.0, type_b.uniform(1E-6), 50, label='d_alpha')
d_theta = ureal(0.0, type_b.uniform(0.05), 2, label='d_theta')

theta_bar = ureal(-0.1,0.2, label='theta_bar')
Delta = ureal(0.0, type_b.arcsine(0.5), label='Delta' )

# Intermediate quantity 'theta'
theta = theta_bar + Delta
```

The length of the standard gauge block is given in a calibration report

```
l_s = ureal(5.0000623E7,25,18,label='ls')
```

two more intermediate results, representing thermal errors, are then

```
# two more intermediate steps
tmp1 = l_s * d_alpha * theta
tmp2 = l_s * alpha_s * d_theta
```

Finally, the length of the gauge block is evaluated

```
# Final equation for the measurement result
l = result( l_s + d - (tmp1 + tmp2), label='l')
```

The script then evaluates the measurement result

```
print( "Measurement result for l={}".format(l) )
```

which displays

```
Measurement result for l=50000838(32)
```

and the following commands display the components of uncertainty for `l`, due to each influence:

```
print("""
Components of uncertainty in l (nm)
-----------------------------------""")

for l_i,u_i in reporting.budget(l):
    print( "  {!s}: {:G}".format(l_i,u_i) )
```

The output is

[3] `ureal` creates a new uncertain real number. It takes a standard uncertainty as its second argument.

```
Components of uncertainty in l (nm)
-----------------------------------
  ls: 25
  d_theta: 16.599
  d2: 6.7
  d0: 5.8
  d1: 3.9
  d_alpha: 2.88679
  alpha_s: 0
  theta_bar: 0
  Delta: 0
```

Resistance and reactance measurement (GUM H2)

An example from Appendix H2 of the GUM[1].

Code

```
from __future__ import print_function
from GTC import *

print("""
-------------------------------
Example from Appendix H2 of GUM
-------------------------------
""")

V = ureal(4.999,3.2E-3,independent=False)      # volt
I = ureal(19.661E-3,9.5E-6,independent=False)  # amp
phi = ureal(1.04446,7.5E-4,independent=False)  # radian

set_correlation(-0.36,V,I)
set_correlation(0.86,V,phi)
set_correlation(-0.65,I,phi)

R = result( V * cos(phi) / I )
X = result( V * sin(phi) / I )
Z = result( V / I )

print('R = {}'.format(R) )
print('X = {}'.format(X) )
print('Z = {}'.format(Z) )
print
print('Correlation between R and X = {:+.2G}'.format( get_correlation(R,X) ) )
print('Correlation between R and Z = {:+.2G}'.format( get_correlation(R,Z) ) )
```

(continues on next page)

[1] BIPM and IEC and IFCC and ISO and IUPAC and IUPAP and OIML, *Evaluation of measurement data - Guide to the expression of uncertainty in measurement JCGM 100:2008 (GUM 1995 with minor corrections)*, (2008) http://www.bipm.org/en/publications/guides/gum

(continued from previous page)

```
print('Correlation between X and Z = {:+.2G}'.format( get_correlation(X,Z) ) )

print("""
(These are not exactly the same values reported in the GUM.
There is some numerical round-off error in the GUM's calculations.)
""")
```

Explanation

Several quantities associated with an electrical component in an AC electrical circuit are of interest here. Measurements of the resistance R, the reactance X and the magnitude of the impedance $|Z|$ are required. These can be obtained by measuring voltage V, current I and phase angle ϕ and then using the measurement equations:

$$R = VI\cos\phi$$
$$X = VI\sin\phi$$
$$|Z| = VI$$

Five repeat measurements of each quantity are performed. The mean values, and associated uncertainties (type-A analysis) provide estimates of voltage, current and phase angle. The correlation coefficients between pairs of estimates is also calculated.

This information is used to define three inputs to the calculation and assign correlation coefficients (the additional argument `independent=False` is required for `set_correlation` to be used).

```
V = ureal(4.999,3.2E-3,independent=False)      # volt
I = ureal(19.661E-3,9.5E-6,independent=False)  # amp
phi = ureal(1.04446,7.5E-4,independent=False)  # radian

set_correlation(-0.36,V,I)
set_correlation(0.86,V,phi)
set_correlation(-0.65,I,phi)
```

Estimates of the three required quantities are then

```
R = result( V * cos(phi) / I )
X = result( V * sin(phi) / I )
Z = result( V / I )
```

Results are displayed by

```
print 'R = {}'.format(R)
print 'X = {}'.format(X)
print 'Z = {}'.format(Z)
print
print 'Correlation between R and X = {:+.2G}'.format( get_correlation(R,X) )
print 'Correlation between R and Z = {:+.2G}'.format( get_correlation(R,Z) )
print 'Correlation between X and Z = {:+.2G}'.format( get_correlation(X,Z) )
```

The output is

```
R = 127.732(70)
X = 219.85(30)
Z = 254.26(24)

Correlation between R and X = -0.59
```

(continues on next page)

(continued from previous page)

```
Correlation between R and Z = -0.49
Correlation between X and Z = +0.99
```

Calculating the expanded uncertainty

The expanded uncertainties for R, X and Z are not evaluated in the GUM, because the Welch-Satterthwaite equation for the effective degrees of freedom is invalid when input estimates are correlated. We created V, I and phi with infinite degrees of freedom, the default).

However, an alternative calculation is applicable in this case[2]. There are two different ways to carry out the calculation in GTC. One uses *type_a.multi_estimate_real()*, the other uses *multiple_ureal()*.

multiple_ureal() creates several elementary uncertain real numbers that are associated with each other (called an *ensemble* in GTC). The documentation shows this applied to the GUM H2 example.

type_a.multi_estimate_real(), performs a type-A analysis on raw data (three sets of five readings) and returns an *ensemble* of elementary uncertain real numbers. The documentation shows this applied to the GUM H2 example.

Note: The impedance calculation can also be treated as a complex-valued problem, so there are other functions that can do data processing of uncertain complex numbers. The documentation for *type_a.multi_estimate_complex()* and *multiple_ucomplex()* both use GUM H2 as an example.

Calibration of a thermometer (GUM H3)

An example from Appendix H3 of the GUM[1].

Code

```
from __future__ import print_function
from GTC import *

print("""
------------------------------
Example from Appendix H3 of GUM
------------------------------
""")
# Thermometer readings (degrees C)
t = (21.521,22.012,22.512,23.003,23.507,23.999,24.513,25.002,25.503,26.010,26.511)
```

(continues on next page)

[2] R Willink, *'A generalization of the Welch-Satterthwaite formula for use with correlated uncertainty components'*, Metrologia **44** (2007) 340-349, Sec. 4.1

[1] BIPM and IEC and IFCC and ISO and IUPAC and IUPAP and OIML, *Evaluation of measurement data - Guide to the expression of uncertainty in measurement JCGM 100:2008 (GUM 1995 with minor corrections)*, (2008) http://www.bipm.org/en/publications/guides/gum

(continued from previous page)

```
# Observed differences with calibration standard (degrees C)
b = (-0.171,-0.169,-0.166,-0.159,-0.164,-0.165,-0.156,-0.157,-0.159,-0.161,-0.160)

# Arbitrary offset temperature (degrees C)
t_0 = 20.0

# Calculate the temperature relative to t_0
t_rel = [ t_k - t_0 for t_k in t ]

# Least-squares regression
cal = type_a.line_fit(t_rel,b)
print( cal )
print

# Apply correction at 30 C
b_30 = cal.intercept + cal.slope*(30.0 - t_0)

print("Correction at 30 C: {}".format(b_30))
```

Explanation

A thermometer is calibrated by comparing 11 readings t_k with corresponding values of a temperature reference standard $t_{\mathrm{R}\cdot k}$.

The readings and the differences $b_k = t_{\mathrm{R}\cdot k} - t_k$ are used to calculate the slope and intercept of a calibration line, which can be used to estimate a temperature correction for a thermometer reading, including the uncertainty.

A linear model of the thermometer is assumed,

$$B_k = Y_1 + Y_2(t_k - t_0) + E_k$$

An arbitrary fixed temperature t_0 is chosen for convenience, t_k is the temperature indicated by the thermometer and B_k is the correction that should be applied to a reading. The constants Y_1 and Y_2 define a linear relationship between the indicated temperature and the correction B_k (Y_1 and Y_2 correspond to the intercept and slope).

The accuracy of the temperature standard is high, so values of $t_{\mathrm{R}\cdot k}$ have no significant error. However, the estimates obtained for the difference between the actual temperature and the indicated temperature are b_k are subject to error.

Estimates of the intercept and slope are obtained by least-squares regression on b_k and t_k. The uncertainty in these estimates is due to random fluctuations in the measurement system (represented by E_k) .

In the GTC calculation, data are contained in a pair of sequences. The function *type_a.line_fit()* performs the regression.

```
# Thermometer readings (degrees C)
t = (21.521,22.012,22.512,23.003,23.507,23.999,24.513,25.002,25.503,26.010,26.511)

# Observed differences with calibration standard (degrees C)
b = (-0.171,-0.169,-0.166,-0.159,-0.164,-0.165,-0.156,-0.157,-0.159,-0.161,-0.160)

# Arbitrary offset temperature (degrees C)
t_0 = 20.0

# Calculate the temperature relative to t_0
t_rel = [ t_k - t_0 for t_k in t ]

# Least-squares regression
```

(continues on next page)

(continued from previous page)

```
cal = type_a.line_fit(t_rel,b)
print cal
print
```

The fitted calibration line can be used to calculate a correction for a reading of 30 C

```
# Apply correction at 30 C
b_30 = cal.intercept + cal.slope*(30.0 - t_0)

print("Correction at 30 C: {}".format(b_30))
```

The results agree with the numbers reported in the GUM

```
-------------------------------
Example from Appendix H3 of GUM
-------------------------------

Ordinary Least-Squares Results:

  Intercept: -0.1712(29)
  Slope: 0.00218(67)
  Correlation: -0.93
  Sum of the squared residuals: 0.000110096583109
  Number of points: 11

Correction at 30 C: -0.1494(41)
```

Other error models

In GUM appendix H.3.6, two alternative scenarios are considered for the thermometer calibration.

Known variance

In the first, the variance of the b_k data is assumed known from prior calibrations.

There are two ways to do this regression problem with GTC.

One way is to define a sequence with the uncertainties of the respective b_k observations. This sequence can be used with *type_a.line_fit_wls()* to obtain the slope and intercept. The other way is to define a sequence of uncertain real numbers representing the b_k data and use the function *type_b.line_fit()*

```
u_b = 0.001 # an arbitrary value, just as an example
cal = type_b.line_fit(t_rel,[ureal(b_i,u_b) for b_i in b])
```

in either case, the results obtained can be used as above to evaluate corrections.

Systematic error in the standard temperature

The other scenario considers a systematic error that causes all b_k values to have some constant offset error. A type-A analysis can still be used on the data to evaluate the contribution to uncertainty due to system instability. However, the systematic error cannot be evaluated by a statistical analysis (it is constant).

This can be handled by combining the results from both type-A and type-B regression analyses.

First, we define a sequence of uncertain real numbers for the b_k data, in which a term representing the systematic error is included

```
E_sys = ureal(0,0.005)
b_sys = [b_i + E_sys for b_i in b]
cal_b = type_b.line_fit(t_rel,b_sys)
print cal_b
```

Note that E_sys, which represents the systematic error, is defined outside the list and then added to each list element. The results are

```
Ordinary Least-Squares Results:

  Intercept: -0.1712(50)
  Slope: 0.00218269773988727725(16)
  Correlation: -1
  Sum of the squared residuals: 0.000110096583109
  Number of points: 11
```

The standard uncertainty in the slope is effectively zero (the small non-zero value can be attributed to numerical round-off error), as expected: an error in the temperature standard shifts all values of b_k by the same amount, so the slope does not change.

Second, a type-A regression analysis is done on the same b_k data sequence (this processes the uncertain-number values, but ignores the uncertainies)

```
cal_a = ta.line_fit(t_rel,b_sys)
print cal_a
```

The results are

```
Ordinary Least-Squares Results:

  Intercept: -0.1712(29)
  Slope: 0.00218(67)
  Correlation: -0.93
  Sum of the squared residuals: 0.000110096583109
  Number of points: 11
```

Notice that the slope and intercept are the same, but not the uncertainties or the correlation coefficient.

In a final step, the results are combined

```
intercept = ta.merge(cal_a.intercept,cal_b.intercept)
slope = ta.merge(cal_a.slope,cal_b.slope)

print( repr(intercept) )
print( repr(slope) )
```

which displays

```
ureal(-0.17120379013135004,0.00576893138292676,145.37964721007157)
ureal(0.002182697739887289 4,0.0006679387732278323,9.0)
```

Notice that neither of the estimates, or the standard uncertainty in the slope, change as a result of merging. However, the standard uncertainty of the intercept does increase, due to uncertainty about the systematic error, as described in H.3.6 in the GUM.

3.1.2 Linear calibration

Linear Regression Results

Conventional least-squares regression of a line to a set of data estimates the parameters of a linear model (slope and intercept) The best-fit line is sometimes called a *calibration line*.

Example

Linear regression is performed with `x` data, that are considered to be error-free stimuli, and `y` data that are observations subject to noise (random errors)

```
>>> x = [3, 7, 11, 15, 18, 27, 29, 30, 30, 31, 31, 32, 33, 33, 34, 36,
...     36, 36, 37, 38, 39, 39, 39, 40, 41, 42, 42, 43, 44, 45, 46, 47, 50]
>>> y = [5, 11, 21, 16, 16, 28, 27, 25, 35, 30, 40, 32, 34, 32, 34, 37,
...     38, 34, 36, 38, 37, 36, 45, 39, 41, 40, 44, 37, 44, 46, 46, 49, 51]

>>> fit = type_a.line_fit(x,y)
```

The object *LineFitOLS* (returned by *type_a.line_fit()*) contains the results of the regression and can be used in different ways.

Estimates of the slope and intercept

Least-squares regression assumes that a model of the system is

$$Y = \alpha + \beta\, x + E\ ,$$

where α and β are unknown parameters, E is a random error with zero mean and unknown variance σ^2, x is the independent (stimulus) variable and Y is the response.

Least-squares regression returns the `fit` object, which holds uncertain numbers representing α and β:

```
>>> a = fit.intercept
>>> a
ureal(3.82963319758869...,1.768447327250652...,31)
>>> b = fit.slope
>>> b
ureal(0.903643210579323...,0.05011897355918200...,31)
>>> get_correlation(a,b)
-0.948124070891915...
```

The response

The uncertain numbers `a` and `b` can be used to estimate the response to a particular stimulus, say $x = 21.5$, in the absence of noise:

```
>>> y = a + 21.5 * b
>>> y
ureal(23.2579622250441...,0.821607058888506...,31.0)
```

The result `y` is an estimate of $\alpha + 21.5 \times \beta$. It is subject to uncertainty because the regression used a sample of data.

A predicted future response

A single future indication in response to a given stimulus may also be of interest. Again, say $x = 21.5$,

```
>>> y0 = fit.y_from_x(21.5)
>>> y0
ureal(23.2579622250441...,3.332409257957110...,31.0)
```

The value here is the same as above (because the stimulus is the same), but the uncertainty is much larger, reflecting the variability of single indications as well as the underlying uncertainty in the intercept and slope.

Estimating the stimulus from observations of the response

Another possibility is that several indications of the response to a steady stimulus are collected. This sample of data may be used to estimate the stimulus[1].

Suppose three observations were collected `[31.4, 29.3, 27.1]`

```
>>> x0 = fit.x_from_y( [31.4, 29.3, 27.1] )
>>> x0
ureal(28.14942133275184...,2.175140873342519...,31.0)
```

`x0` is an estimate of the stimulus based on the observations, but also taking into account the variability in the `y` data used earlier in the regression.

Straight-line calibration functions

[1] This scenario is sometimes called *calibration*. The response of an instrument to a number of different reference stimuli is observed and a calibration curve is calculated. The curve is then used in the opposite sense, to convert observations of the instrument response into estimates of the stimulus applied.

- *Application: an additional response*
- *Forward evaluation: an additional stimulus*

This section uses straight-line least-squares regression algorithms to obtain calibration functions and then shows some uses of those functions. Each example uses a small sample of x-y observation pairs for regression to obtain a calibration function, together with information about the variability of the data. No context is given about the measurement; these examples have been selected from a draft British standard on the use of straight-line calibration functions[1].

In some examples, we show how results can be used to estimate a stimulus value `x`, when given additional observations of the response `y`, or a future response `y` to a given stimulus `x`.

Example 1: equal weights

A series of six pairs of `x`-`y` observations have been collected.

The data sequences `x` and `y` and a sequence of uncertainties in the y values are

```
>>> x = [1,2,3,4,5,6]
>>> y = [3.3,5.6,7.1,9.3,10.7,12.1]
>>> u_y = [0.5] * 6
```

We apply weighted least-squares regression to the data, which assumes that the values in `u_y` are known standard deviations for noise in `y` data (i.e., the data have infinite degrees of freedom)

```
>>> fit = type_a.line_fit_wls(x,y,u_y)
>>> print(fit)

Weighted Least-Squares Results:

  Intercept:  1.87(47)
  Slope:  1.76(12)
  Correlation: -0.9
  Sum of the squared residuals: 1.66476190476...
  Number of points: 6
```

More significant figures could be obtained with

```
>>> a = fit.intercept
>>> print( "a={:.15G}, u={:.15G}".format(value(a),uncertainty(a)) )
a=1.86666666666667, u=0.465474668125631
>>> b = fit.slope
>>> print( "b={:.15G}, u={:.15G}".format(value(b),uncertainty(b)) )
b=1.75714285714286, u=0.119522860933439
>>> print( "cov(a,b)={:.15G}".format(a.u*b.u*get_correlation(a,b)) )
cov(a,b)=-0.05
```

These results agree with published values[2]

```
a = 1.867, u(a) = 0.465
b = 1.757, u(b) = 0.120
cov(a,b) = -0.050
chi-squared = 1.665, with 4 degrees of freedom
```

[1] These examples also appear in BS DD ISO/TS 28037:2010 *Determination and use of straight-line calibration functions*, (British Standards Institute, 2010).

[2] Section 6.3, page 13, in BS DD ISO/TS 28037:2010.

The value of `chi-squared` can be compared with the `Sum of the squared residuals` above and the degrees of freedom is the `Number of points` less 2.

Application: an additional *y* observation after regression

The results may be used to find a value for `x` that corresponds to another observation `y` made following the regression. This is a typical application of a calibration curve.

For example, if an additional observation $y_1 = 10.5$ has been made, with $u(y_1) = 0.5$, we can evaluate an uncertain number for the corresponding stimulus x_1:

```
>>> y1 = ureal(10.5,0.5)
>>> x1 = (y1-a)/b
>>> print( "x1={:.15G}, u={:.15G}".format(value(x1),uncertainty(x1)) )
x1=4.91327913279133, u=0.322035560128 91
```

This uncertain number has components of uncertainty for the estimates of slope and intercept. So the combined uncertainty takes account of uncertainty in the parameter estimates for the calibration curve, and correlation between them.

Forward evaluation: an additional *x* value

The results can also be used to estimate the response y_2 to a stimulus x_2.

For example, if $x_2 = 3.5$, and $u(x_2) = 0.2$, we can evaluate an uncertain number for y_2 as follows

```
>>> x2 = ureal(3.5,0.2)
>>> y2 = a + b*x2
>>> print( "y2={:.15G}, u={:.15G}".format(value(y2),uncertainty(y2)) )
y2=8.01666666666667, u=0.406409531732455
```

This is an uncertain number representing the mean, or underlying true, response to x_2. Again, the uncertain number for y_2 has components of uncertainty for the estimates of slope and intercept.

Example 2: unequal weights

A series of six pairs of `x`-`y` observations have been collected.

The data sequences for `x` and `y`, with uncertainties in `y`, are

```
>>> x = [1,2,3,4,5,6]
>>> y = [3.2, 4.3, 7.6, 8.6, 11.7, 12.8]
>>> u_y = [0.5,0.5,0.5,1.0,1.0,1.0]
```

Again, a weighted least-squares regression can be used, which assumes that the uncertainties in `y` values are exactly known (i.e., infinite degrees of freedom)

```
>>> fit = type_a.line_fit_wls(x,y,u_y)
>>> print( fit )

Weighted Least-Squares Results:

  Intercept:  0.89(53)
  Slope:  2.06(18)
  Correlation: -0.87
  Sum of the squared residuals: 4.13080168776...
  Number of points: 6
```

More significant figures can be obtained by the same commands used in Example 1:

```
a=0.885232067510549, u=0.529708143508836
b=2.05696202531646, u=0.177892016741205
cov(a,b)=-0.0822784810126582
```

These results agree with published values[3]

```
a = 0.885, u(a) = 0.530
b = 2.057, u(b) = 0.178
cov(a,b) = -0.082
chi-squared = 4.131, with 4 degrees of freedom
```

Application: an additional *y* observation after regression

After regression, the uncertain numbers for the intercept and slope can be used to estimate the stimulus x_1 for a further observation y_1. For example, if $y_1 = 10.5$ and $u(y_1) = 1.0$, x_1 is obtained in the same way as Example 1

```
>>> a = fit.intercept
>>> b = fit.slope
>>> y1 = ureal(10.5,1)
>>> x1 = (y1 - a)/b
>>> print( "x1={:.15G}, u={:.15G}".format( value(x1),uncertainty(x1) ) )
x1=4.674..., u=0.533...
```

Example 3: uncertainty in *x* and *y*

A series of six pairs of observations have been collected.

The data sequences for x, y, each with uncertainties are

```
>>> x = [1.2,1.9,2.9,4.0,4.7,5.9]
>>> u_x = [0.2] * 6
>>> y = [3.4,4.4,7.2,8.5,10.8,13.5]
>>> u_y = [0.2,0.2,0.2,0.4,0.4,0.4]
```

We use total least-squares regression in this case, because there is uncertainty in both the dependent and independent variablest

```
>>> fit = type_a.line_fit_wtls(x,y,u_x,u_y,fit.a_b)
>>> print( fit )

Weighted Total Least-Squares Results:

  Intercept:  0.58(48)
  Slope:  2.16(14)
  Correlation: -0.9
  Sum of the squared residuals: 2.74267678973...
  Number of points: 6
```

Again, more figures can be obtained using the same commands as in Example 1

```
a=0.578822122145264, u=0.480359046511757
b=2.15965656740064, u=0.136246483136605
cov(a,b)=-0.0586143419560877
```

[3] Section 6.3, page 15, in BS DD ISO/TS 28037:2010.

These results agree with the published values[4]

```
a = 0.5788, u(a) = 0.0.4764
b = 2.159, u(b) = 0.1355
cov(a,b) = -0.0577
chi-squared = 2.743, with 4 degrees of freedom
```

(There are slight differences due to a different number of iterations in the TLS calculation.)

Example 4: relative uncertainty in *y*

A series of six pairs of `x-y` observations are used. The uncertainties in the y values are not known. However, a scale factor s_y is given and it is assumed that, for every observation y, the associated uncertainty $u(y) = s_y\sigma$. The common factor σ is not known, but can be estimated from the residuals. This is done by the function *type_a.line_fit_rwls()*.

We proceed as above

```
>>> x = [1,2,3,4,5,6]
>>> y = [3.014,5.225,7.004,9.061,11.201,12.762]
>>> u_y = [1] * 6
>>> fit = type_a.line_fit_rwls(x,y,u_y)
>>> print( fit )

Relative Weighted Least-Squares Results:

  Intercept:  1.17(16)
  Slope:  1.964(41)
  Correlation: -0.9
  Sum of the squared residuals: 0.11649828571...
  Number of points: 6
```

More precise values of the fitted parameters are

```
a=1.172, u=0.158875093196181
b=1.96357142857143, u=0.0407953578791729
cov(a,b)=-0.00582491428571429
```

These results agree with the published values[5]

```
a = 1.172, u(a) = 0.159
b = 1.964, u(b) = 0.041
cov(a,b) = -0.006
chi-squared = 0.171, with 4 degrees of freedom
```

Note: In our solution, 4 degrees of freedom are associated with estimates of the intercept and slope. This is the usual statistical treatment. However, a trend in recent uncertainty guidelines is to dispense with the notion of degrees of freedom. So, in a final step, reference[Page 74, 1] multiplies $u(a)$ and $u(b)$ by an additional factor of 2. We do not agree with this last step. GTC uses the finite degrees of freedom associated with the intercept and slope to calculate the coverage factor required for an expanded uncertainty.

---

[4] Section 7.4, page 21, in BS DD ISO/TS 28037:2010.
[5] Appendix E, pages 58-59, in BS DD ISO/TS 28037:2010.

Example 5: unknown uncertainty in *y*

The data in previous example could also have been processed by an 'ordinary' least-squares regression algorithm, because the scale factor for each observation of y was unity. In effect, a series of six values for the dependent and independent variables were collected, and the variance associated with each observation was assumed to be the same.

We proceed as follows. The data sequences are defined and the ordinary least-squares function is applied

```
>>> x = [1,2,3,4,5,6]
>>> y = [3.014,5.225,7.004,9.061,11.201,12.762]
>>> fit = type_a.line_fit(x,y)
>>> print( fit )

Ordinary Least-Squares Results:

  Intercept:  1.17(16)
  Slope:  1.964(41)
  Correlation: -0.9
  Sum of the squared residuals: 0.11649828571...
  Number of points: 6
```

More precise values of the fitted parameters are

```
a=1.172, u=0.158875093196181
b=1.96357142857143, u=0.0407953578791729
cov(a,b)=-0.00582491428571429
```

The same results were obtained in Example 4.

Application: an additional response

After regression, if a further observation of y becomes available, or a set of observations, then the corresponding stimulus can be estimated.

For example, if we wish to know the stimulus x_1 that gave rise to a response $y_1 = 10.5$, we can use the object `fit` returned by the regression (note that *x_from_y()* takes a sequence of y values)

```
>>> y1 = 10.5
>>> x1 = fit.x_from_y( [y1] )
>>> print( x1 )
4.751(97)
```

Forward evaluation: an additional stimulus

The regression results can also be used to predict a single future response y for a given stimulus x.

For example, if $x_2 = 3.5$ we can find y_2 as follows

```
>>> x2 = 3.5
>>> y2 = fit.y_from_x(x2)
>>> print( y2 )
8.04(18)
```

In this case, the uncertainty reported for y_2 includes a component for the variability of individual responses. The method *y_from_x()* incorporates this information from the regression analysis.

Alternatively, the mean response to a stimulus x_2 can be obtained directly from the fitted parameters

```
>>> x2 = 3.5
>>> a, b = fit.a_b
>>> y2 = a + b*x2
>>> print( y2 )
8.044(70)
```

Linear Calibration Equations

This section applies GTC to a simple calibration problem[1].

Calibration

A pressure sensor with an approximately linear response is to be calibrated.

Eleven reference pressures are accurately generated and the corresponding sensor indications are recorded. The standard pressure values are entered in the`y_data` sequence and sensor readings in *x_data* (data from Table 4 in[1]):

```
>>> y_data = (0.0,2.0,4.0,6.0,8.0,10.0,12.0,14.0,16.0,18.0,20.0)
>>> x_data = (0.0000,0.2039,0.4080,0.6120,0.8160,1.0201,1.2242,1.4283,1.6325,1.8367,2.
↪0410)
```

The sensor indication does not change when observations are repeated at the same reference pressure values, which suggests that the digital resolution of the sensor is much less than any repeatability errors associated with calibration. So we ignore random noise as a source of error.

Measurement model

A linear model of the sensor's behaviour is

$$Y = \alpha + \beta\, X \ ,$$

where Y represents the applied pressure and X the sensor response.

In operation, the sensor indication, x is taken as an estimate of X. The relationship between an applied pressure Y_i and the indication x_i may be expressed as

$$Y_i = \alpha + \beta\,(x_i - E_{\mathrm{res}\cdot i}) + E_{\mathrm{lin}\cdot i}$$

where $E_{\mathrm{res}\cdot i}$ and $E_{\mathrm{lin}\cdot i}$ are errors.

$E_{\mathrm{res}\cdot i}$ is a round-off error due to the finite number of digits displayed (i.e., instead of X_i, the number displayed is $x_i = X_i + E_{\mathrm{res}\cdot i}$).

[1] R Kessel, R N Kacker and K-D Sommer, *Uncertainty budget for range calibration*, Measurement **45** (2012) 1661 – 1669.

$E_{\mathrm{lin}\cdot i}$ is the difference between an actual applied pressure Y_i and the pressure predicted by the linear model $\alpha+\beta\, X_i$.

During calibration, the applied reference pressure $Y_{\mathrm{cal}\cdot i}$ is not known exactly. The nominal reference pressure is

$$y_{\mathrm{cal}\cdot i} = Y_{\mathrm{cal}\cdot i} + E_{\mathrm{cal}\cdot i} \ ,$$

where $E_{\mathrm{cal}\cdot i}$ is a measurement error in the reference. The uncertainty of $y_{\mathrm{cal}\cdot i}$ as an estimate of $Y_{\mathrm{cal}\cdot i}$ is given as a relative standard uncertainty

$$\frac{u(y_{\mathrm{cal}\cdot i})}{y_{\mathrm{cal}\cdot i}} = 0.000115 \ .$$

A 2-point calibration curve

A calibration procedure estimates α and β. The actual slope, β, is

$$\beta = \frac{Y_{\mathrm{cal}\cdot 10} - Y_{\mathrm{cal}\cdot 0}}{X_{\mathrm{cal}\cdot 10} - X_{\mathrm{cal}\cdot 0}} \ .$$

Points near the ends of the range of data available are most influential when estimating the slope and intercept of a linear calibration function, So, an estimate of the slope is

$$b = \frac{y_{\mathrm{cal}\cdot 10} - y_{\mathrm{cal}\cdot 0}}{x_{\mathrm{cal}\cdot 10} - x_{\mathrm{cal}\cdot 0}} \ .$$

Using uncertain numbers, this can be calculated

```
>>> u_ycal_rel = 0.000115
>>> u_res = type_b.uniform(0.00005)

>>> x_0 = x_data[0] - ureal(0,u_res,label='e_res_0')
>>> x_10 = x_data[10] - ureal(0,u_res,label='e_res_10')

>>> y_0 = ureal(y_data[0],y_data[0]*u_ycal_rel,label='y_0')
>>> y_10 = ureal(y_data[10],y_data[10]*u_ycal_rel,label='y_10')

>>> b = (y_10 - y_0)/(x_10 - x_0)
>>> a = y_10 - b * x_10
```

The results for `a` and `b`, as well as the correlation coefficient, are

```
>>> a
ureal(0.0,0.0002828761730473424,inf)
>>> b
ureal(9.799118079372857,0.0011438175474686209,inf)
>>> get_correlation(a,b)
-0.12117041864179227
```

The non-linearity error

Using the remainder of the calibration data, we can compare the calibration line with the calibration data points and thereby assess the importance of non-linear sensor response across the range. The following will display a table of differences between the data and the model

```
>>> for x_i,y_i in zip(x_data,y_data):
...     dif = y_i - (x_i * b + a)
...     print("x = {:G}, dif ={}".format(x_i,dif))
...
```

(continues on next page)

(continued from previous page)

```
x = 0, dif =...0.00000(28)
x = 0.2039, dif = 0.00196(34)
x = 0.408, dif = 0.00196(52)
x = 0.612, dif = 0.00294(72)
x = 0.816, dif = 0.00392(94)
x = 1.0201, dif = 0.0039(12)
x = 1.2242, dif = 0.0039(14)
x = 1.4283, dif = 0.0039(16)
x = 1.6325, dif = 0.0029(19)
x = 1.8367, dif = 0.0020(21)
x = 2.041, dif = 0.0000(23)
```

A maximum deviation (worst case error) is taken to be 0.005.[#Kessel]_ This amount of deviation is assumed to cover departures from linearity of the sensor[2].

The calibration equation

We now have sufficient information to define a calibration function that takes a sensor indication and returns an uncertain number for applied pressure. For instance,

```
>>> u_lin = type_b.uniform(0.005)
>>> u_res = type_b.uniform(0.00005)

>>> a = ureal(0.0,0.00028,label='a',independent=False)
>>> b = ureal(9.79912, 0.00114,label='b',independent=False)
>>> set_correlation(-0.1212,a,b)

>>> def cal_fn(x):
...     """-> pressure estimate
...     :arg x: sensor reading (a number)
...     :returns: an uncertain number representing the applied pressure
...     """
...     e_res_i = ureal(0,u_res,label='e_res_i')
...     e_lin_i = ureal(0,u_lin,label='e_lin_i')
...     return a + b * (x + e_res_i) + e_lin_i
...
```

With this function, we can calculate pressures and expanded uncertainties ($k = 2$) for the calibration data, which can be compared with Table 7 in the reference[Page 79, 1]

```
>>> for i,x_i in enumerate(x_data):
...     y_i = cal_fn(x_i)
...     print("{}: p={:G},  U(p)={:G}".format(i,y_i.x,2*y_i.u))
...
0: p=0,  U(p)=0.00582812
1: p=1.99804,  U(p)=0.00584124
2: p=3.99804,  U(p)=0.00589119
3: p=5.99706,  U(p)=0.00597701
4: p=7.99608,  U(p)=0.0060972
5: p=9.99608,  U(p)=0.00624986
6: p=11.9961,  U(p)=0.00643263
7: p=13.9961,  U(p)=0.00664303
8: p=15.9971,  U(p)=0.00687865
```

(continues on next page)

[2] The uncertainty due to linearity errors can be estimated later by comparing the calibration data with the pressure predicted by the linear calibration curve.

(continued from previous page)

```
9: p=17.998,  U(p)=0.00713689
10: p=20,  U(p)=0.00741554
```

Linearising the sensor response

With additional information about the typical behaviour of this type of sensor, we can pre-process readings and improve the linearity of the response. The following equation takes a raw indication x and returns a value that will vary more linearly with applied pressure than x. The effect of f_{lin} is to reduce the difference between the pressure estimates and actual pressures.

$$f_{\text{lin}}(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3$$

The coefficients c_i apply to the type of sensor; they are **not** determined as part of the calibration procedure. No uncertainty need be associated with these numbers.

The pre-processing function can be implemented as

```
>>> def f_lin(x):
...     """improve sensor linearity"""
...     c0 = 0.0
...     c1 = 9.806
...     c2 = -2.251E-3
...     c3 = -5.753E-4
...     return c0 + (c1 + (c2 + c3*x)*x)*x
...
```

Our model of the measurement is now

$$Y_i = \alpha + \beta\, f_{\text{lin}}(x_i - E_{\text{res}\cdot i}) + E_{\text{lin}\cdot i}$$

To calibrate this 'linearised' sensor, the original indications $x_{\text{cal}\cdot 10}$ and $x_{\text{cal}\cdot 0}$ are transformed by $f_{\text{lin}}(X)$ before calculating the slope and intercept (this transformation also takes account of the reading error).

```
>>> u_ycal_rel = 0.000115
>>> u_res = type_b.uniform(0.00005)

>>> x_0 = f_lin( x_data[0] - ureal(0,u_res,label='e_res_0') )
>>> x_10 = f_lin( x_data[10] - ureal(0,u_res,label='e_res_10') )

>>> y_0 = ureal(y_data[0],y_data[0]*u_ycal_rel,label='y_0')
>>> y_10 = ureal(y_data[10],y_data[10]*u_ycal_rel,label='y_10')

>>> b = (y_10 - y_0)/(x_10 - x_0)
>>> a = y_10 - b * x_10
```

The results are

```
>>> a
ureal(0.0,0.0002830779825130 5335,inf)
>>> b
ureal(1.000011112006328,0.0001167274598608 2041,inf)
>>> get_correlation(a,b)
-0.12125729816056871
```

The differences between nominal standard values and the sensor estimates can be displayed by

```
>>> for x_i,y_i in zip(x_data,y_data):
...     dif = y_i - (f_lin(x_i) * b + a)
...     print("x = {: G}, dif ={}".format(x_i,dif))
...
x = 0, dif =...0.00000(28)
x = 0.2039, dif = 0.00063(34)
x = 0.408, dif =-0.00048(52)
x = 0.612, dif =-0.00036(72)
x = 0.816, dif = 0.00003(94)
x = 1.0201, dif =-0.0003(12)
x = 1.2242, dif =-0.0002(14)
x = 1.4283, dif = 0.0002(16)
x = 1.6325, dif = 0.0000(19)
x = 1.8367, dif = 0.0003(21)
x = 2.041, dif = 0.0000(23)
```

Which shows that the differences are much smaller than before.

The worst-case error is now about ± 0.0007.

The new calibration equation

A new calibration function that takes a sensor indication and returns the applied pressure can be defined

```
>>> u_lin = type_b.uniform(0.0007)
>>> u_res = type_b.uniform(0.00005)

>>> a = ureal(0.0,0.00028,label='a',independent=False)
>>> b = ureal(1.000011, 0.000117,label='b',independent=False)
>>> set_correlation(-0.1215,a,b)

>>> def lin_cal_fn(x):
...     """-> linearised pressure estimate
...     :arg x: sensor reading (a number)
...     :returns: an uncertain number representing the applied pressure
...     """
...     e_res_i = ureal(0,u_res,label='e_res_i')
...     e_lin_i = ureal(0,u_lin,label='e_lin_i')
...     return a + b * f_lin(x + e_res_i) + e_lin_i
...
```

The improvement to accuracy can be seen by applying this function to the calibration data

```
>>> for i,x_i in enumerate(x_data):
...     y_i = lin_cal_fn(x_i)
...     print("{}: p={:0.5G},  U(p)={:.2G}".format(i,y_i.x,2*y_i.u))
...
0: p=0,  U(p)=0.0011
1: p=1.9994,  U(p)=0.0012
2: p=4.0005,  U(p)=0.0014
3: p=6.0004,  U(p)=0.0018
4: p=8,  U(p)=0.0021
5: p=10,  U(p)=0.0025
6: p=12,  U(p)=0.003
7: p=14,  U(p)=0.0034
8: p=16,  U(p)=0.0038
```

(continues on next page)

(continued from previous page)

```
9: p=18,  U(p)=0.0043
10: p=20,  U(p)=0.0047
```

3.1.3 EURACHEM / CITAC Guide Examples

Preparation of a Calibration Standard (A1)

This section is based on a measurement described in Appendix 1 of the 3rd edition of the EURACHEM / CITAC Guide[1].

The CITAC Guide gives a careful discussion of the uncertainty analysis leading to particular numerical values. The following shows only how the subsequent calculation can be preformed using GTC.

The measurement

The concentration of Cd in a standard solution is to be determined.

This can be expressed by the equation

$$c_{\mathrm{Cd}} = \frac{1000 \cdot m \cdot P}{V} \ ,$$

where

- c_{Cd} is the concentration expressed (mg/L),
- 1000 is a conversion factor from mL to L,
- m is the mass of high purity metal (mg),
- P is the purity of the metal as a mass fraction,
- V is the volume of liquid of the standard (mL).

The uncertainty contributions

In section A1.4 of the CITAC Guide the numerical estimates of influence quantities are described. These can be used to define uncertain numbers for the mass, purity and volume. The mass and purity are defined directly as elementary uncertain numbers[2]:

```
>>> P = ureal(0.9999,type_b.uniform(0.0001),label='P')
>>> m = ureal(100.28,0.05,label='m') # mg
```

The volume has three influences that contribute to the overall uncertainty: the manufacturing tolerances of the measuring flask, the repeatability of filling and the variability of temperature during the experiment. Each is represented by an elementary uncertain number

```
>>> V_flask = ureal(100,type_b.triangular(0.1),label='V_flask')
>>> V_rep = ureal(0,0.02,label='V_rep')
>>> V_T = ureal(0,type_b.uniform(0.084),label='V_T')
```

Note that the value assigned to `V_rep` and `V_T` is zero. These represent repeatability error and the temperature error incurred during the experiment. The best estimate of these errors is zero but the uncertainty is given in the second argument to `ureal`.

After these definitions an uncertain number representing the volume of fluid is (we label the uncertain number for convenience when reporting the uncertainty budget later)

[1] On-line: http://www.citac.cc/QUAM2012_P1.pdf

[2] Functions from the *type_b* module are used to scale the uncertainty parameter of a non-Gaussian error to obtain the standard deviation.

```
>>> V = result( V_flask + V_rep + V_T,label = 'V')
```

The uncertainty calculation

The concentration calculation is then simply[3]

```
>>> c_Cd = 1000 * m * P / V
>>> print( "c_Cd={:G}, u={:G}".format(c_Cd.x,c_Cd.u) )
c_Cd=1002.7, u=0.835199
```

The contributions to the standard uncertainty can be itemised using *reporting.budget()*:

```
>>> for i in rp.budget(c_Cd):
...     print( " {}: {:G}".format(i.label,i.u) )
...
 m: 0.49995
 V_T: 0.486284
 V_flask: 0.40935
 V_rep: 0.20054
 P: 0.0578967
```

The contribution from the overall uncertainty in the volume of fluid, rather than the individual terms can also be compared with other contributions by using a list of influences

```
>>> for i in rp.budget(c_Cd,influences=[m,P,V]):
...     print( " {}: {:G}".format(i.label,i.u) )
...
 V: 0.666525
 m: 0.49995
 P: 0.0578967
```

These results can be compared with Figure A1.5 in the CITAC Guide.

Standardising a Sodium Hydroxide Solution (A2)

This section is based on a measurement described in Appendix 2 of the 3rd edition of the EURACHEM / CITAC Guide[1].

The CITAC Guide gives a careful discussion of the uncertainty analysis leading to particular numerical values. The following shows only how the subsequent calculation can be preformed using GTC.

The measurement

The concentration of a solution of NaOH is to be determined. The NaOH is titrated against the titrimetric standard potassium hydrogen phthalate (KHP).

The measurand can be expressed as

$$c_{\mathrm{NaOH}} = \frac{1000 \cdot m_{\mathrm{KHP}} \cdot P_{\mathrm{KHP}}}{M_{\mathrm{KHP}} \cdot V_{\mathrm{T}}} \ ,$$

where

- c_{NaOH} is the concentration expressed in mol/L,

[3] The numbers differ slightly because numbers in the CITAC Guide calculations have been rounded
[1] On-line: http://www.citac.cc/QUAM2012_P1.pdf

- 1000 is a volume conversion factor from mL to L,
- m_{KHP} is the mass of the titrimetric standard in g,
- P_{KHP} is the purity of the titrimetric standard as a mass fraction,
- M_{KHP} is the molar mass of KHP in g/mol,
- V_{T} is the titration volume of NaOH solution in mL.

The uncertainty contributions

Section A2.4 of the CITAC Guide provides numerical estimates of influence quantities, which can be used to define uncertain numbers for the calculation.

The mass m_{KHP} is determined from the difference of two weighings with balance linearity as the only source of measurement error considered. However, a linearity error occurs twice: once in the tare weighing and once in the gross weighing. So in the calculations we introduce the nett weight as a number (0.3888) and the uncertainty contribution is found by taking the difference of uncertain numbers representing the errors that occur during the weighings (if the raw observations were available, they might have been used to define `u_lin_tare` and `u_lin_gross`)[2].

```
>>> u_lin_tare = ureal(0,type_b.uniform(0.15E-3),label='u_lin_tare')
>>> u_lin_gross = ureal(0,type_b.uniform(0.15E-3),label='u_lin_gross')
>>> u_m_KHP = u_lin_gross - u_lin_tare
>>> m_KHP = result( 0.3888 + u_m_KHP, label='m_KHP' )
```

The purity P_{KHP} is[3]

```
>>> P_KHP = ureal(1.0,type_b.uniform(0.0005),label='P_KHP')
```

The molar mass m_{KHP} is calculated from IUPAC data and the number of each constituent element in the KHP molecule $\mathrm{C_8H_5O_4K}$.

```
>>> M_C = ureal(12.0107,type_b.uniform(0.0008),label='M_C')
>>> M_H = ureal(1.00794,type_b.uniform(0.00007),label='M_H')
>>> M_O = ureal(15.9994,type_b.uniform(0.0003),label='M_O')
>>> M_K = ureal(39.0983,type_b.uniform(0.0001),label='M_K')

>>> M_KHP = result( 8*M_C + 5*M_H + 4*M_O + M_K, label='M_KHP' )
```

The volume term V_{T2} is affected by contributions from calibration error and temperature.

```
>>> uV_T_cal = ureal(0,type_b.triangular(0.03),label='V_T_cal')
>>> uV_T_temp = ureal(0,0.006,label='V_T_temp')

>>> V_T = result( 18.64 + uV_T_cal + uV_T_temp, label='V_T' )
```

The CITAC Guide introduces a further multiplicative term R to represent repeatability errors ($R \approx 1$)

$$c_{\mathrm{NaOH}} = R\,\frac{1000 \cdot m_{\mathrm{KHP}} \cdot P_{\mathrm{KHP}}}{M_{\mathrm{KHP}} \cdot V_{\mathrm{T}}}\ ,$$

In the GTC calculation this is represented by another uncertain number

```
>>> R = ureal(1.0,0.0005,label='R')
```

[2] If the balance indications for the tare and gross weighings were known they could have been used to define the values of these uncertain numbers, however the Guide does not provide this raw data. Instead, the zero value used here represents an estimate of the linearity *error*.

[3] Functions from the *type_b* module are used here to scale the uncertainty parameters, as described in the CITAC Guide

The uncertainty calculation

The calculation of c_{NaOH} is now[4]:

```
>>> c_NaOH = R * (1000 * m_KHP * P_KHP)/(M_KHP * V_T)
>>> c_NaOH
ureal(0.102136159706...,0.000100500722124...,inf)
```

The contribution from different influences can be examined (and compared with Fig. A2.9 in the Guide)

```
>>> for i in rp.budget(c_NaOH,influences=[m_KHP,P_KHP,M_KHP,V_T,R]):
...         print( " {}: {:G}".format(i.label,i.u) )
...
 V_T: 7.47292E-05
 R: 5.10681E-05
 m_KHP: 3.21735E-05
 P_KHP: 2.94842E-05
 M_KHP: 1.88312E-06
```

The full uncertainty budget is

```
>>> for i in rp.budget(c_NaOH):
...         print( " {}: {:G}".format(i.label,i.u) )
...
 V_T_cal: 6.71088E-05
 R: 5.10681E-05
 V_T_temp: 3.28764E-05
 P_KHP: 2.94842E-05
 u_lin_tare: 2.27501E-05
 u_lin_gross: 2.27501E-05
 M_C: 1.84798E-06
```

An Acid/Base Titration (A3)

This section is based on a measurement described in Appendix Appendix 3 of the 3rd edition of the EURACHEM / CITAC Guide[1].

The CITAC Guide gives a careful discussion of the uncertainty analysis leading to particular numerical values. The following shows only how the subsequent calculation can be preformed using GTC.

The measurement

The method determines the concentration of an HCl solution by a sequence of experiments. This is a longer calculation than the previous examples, so the code shown below should be considered as lines of text in a file that can be executed by GTC.

The measurand can be expressed by

$$c_{\mathrm{HCl}} = \frac{1000 \cdot m_{\mathrm{KHP}} \cdot P_{\mathrm{KHP}} \cdot V_{\mathrm{T2}}}{V_{\mathrm{T1}} \cdot M_{\mathrm{KHP}} \cdot V_{\mathrm{HCl}}} \ ,$$

where

- c_{HCl} is the concentration expressed (mol/L),
- 1000 is a volume conversion factor from mL to L,

[4] The numbers differ slightly because numbers in the the CITAC Guide calculations have been rounded

[1] On-line: http://www.citac.cc/QUAM2012_P1.pdf

- m_{KHP} is the mass of KHP taken (g),
- P_{KHP} is the purity of KHP as a mass fraction,
- V_{T1} is the volume of NaOH to titrate KHP (mL).
- V_{T2} is the volume of NaOH to titrate HCl (mL).
- M_{KHP} is the molar mass of KHP (g/mol),
- V_{T} is the titration volume of NaOH solution (mL).

The uncertainty contributions

Section A3.4 of the CITAC Guide provides numerical estimates of influence quantities, which can be used to define uncertain numbers for the uncertainty calculation.

The mass m_{KHP} is determined from the difference of two weighings with balance linearity as the only source of measurement error. However, a linearity error arises twice: once in the tare weighing and once in the gross weighing. So, in the calculations we introduce the nett weight as a number (0.3888) and the uncertainty contribution is found by taking the difference of uncertain numbers representing estimates of the errors that occur during the weighings (if the raw observations were available, they might have been used to define `u_lin_tare` and `u_lin_gross`)[2].

```
>>> u_lin_tare = ureal(0,type_b.uniform(0.15E-3),label='u_lin_tare')
>>> u_lin_gross = ureal(0,type_b.uniform(0.15E-3),label='u_lin_gross')
>>> m_KHP = result( 0.3888 + u_lin_gross - u_lin_tare, label = 'm_KHP' )
```

The purity P_{KHP} is[3]

```
>>> P_KHP = ureal(1.0,type_b.uniform(0.0005),label='P_KHP')
```

The volume term V_{T2} is affected by contributions from calibration error and temperature. In calculating the uncertainty contribution due to temperature, the volume expansion coefficient for water $2.1 \times 10^{-4}\,^{\circ}\mathrm{C}^{-1}$ is used, the volume of the pipette is 15 mL and the temperature range is $\pm 4\,^{\circ}\mathrm{C}$.

```
>>> uV_T2_cal = ureal(0,type_b.triangular(0.03),label='V_T2_cal')
>>> uV_T2_temp = ureal(0,type_b.uniform(15 * 2.1E-4 * 4),label='V_T2_temp')

>>> V_T2 = result( 14.89 + uV_T2_cal + uV_T2_temp, label='V_T2' )
```

The influences of the volume term V_{T1} are almost the same as V_{T2}, only the temperature contribution is different because a 19 mL volume of NaOH was used.

```
>>> uV_T1_cal = ureal(0,type_b.triangular(0.03),label='V_T1_cal')
>>> uV_T1_temp = ureal(0,type_b.uniform(19 * 2.1E-4 * 4),label='V_T1_temp')

>>> V_T1 = result( 18.64 + uV_T1_cal + uV_T1_temp, label = 'V_T1' )
```

The molar mass m_{KHP} is calculated from IUPAC data and the number of each constituent element in the KHP molecule $\mathrm{C_8H_5O_4K}$. This can be done as follows

```
>>> M_C = ureal(12.0107,type_b.uniform(0.0008),label='M_C')
>>> M_H = ureal(1.00794,type_b.uniform(0.00007),label='M_H')
>>> M_O = ureal(15.9994,type_b.uniform(0.0003),label='M_O')
>>> M_K = ureal(39.0983,type_b.uniform(0.0001),label='M_K')

>>> M_KHP = result( 8*M_C + 5*M_H + 4*M_O + M_K, label='M_KHP' )
```

[2] If the balance indications for the tare weighing and gross weighing were known they could have been used to define the values of these uncertain numbers, however the CITAC Guide does not provide this raw data. Instead, the zero value used here represents the linearity *error*.

[3] Functions from the *type_b* module are used here to scale the uncertainty parameters, as described in the CITAC Guide

The influences on the volume term V_{HCl} are similar to the V_{T1} and V_{T2}. A 15 mL pipette was used with a stated uncertainty tolerance of 0.02. The range of temperature variation in the laboratory is $4\,^{\circ}\mathrm{C}$.

```
>>> uV_HCl_cal = ureal(0,type_b.triangular(0.02),label='uV_HCl_cal')
>>> uV_HCl_temp = ureal(0,type_b.uniform(15 * 2.1E-4 * 4),label='uV_HCl_temp')

>>> V_HCl = result( 15 + uV_HCl_cal + uV_HCl_temp, label='V_HCl' )
```

The CITAC Guide introduces a further multiplicative term R to represent repeatability error ($R \approx 1$)

$$c_{\mathrm{NaOH}} = R\,\frac{1000 \cdot m_{\mathrm{KHP}} \cdot P_{\mathrm{KHP}}}{M_{\mathrm{KHP}} \cdot V_{\mathrm{T}}}\ ,$$

Another uncertain number is defined to represent this

```
>>> R = ureal(1.0,0.001,label='R')
```

The uncertainty calculation

The calculation of c_{NaOH} is now

```
>>> c_HCl = R * (1000 * m_KHP * P_KHP * V_T2)/(M_KHP * V_T1 * V_HCl)
>>> print(c_HCl)
0.10139(18)
>>> print( uncertainty(c_HCl) )
0.0001843...
```

The contribution from different influences can be examined

```
>>> for i in rp.budget(c_HCl,influences=[m_KHP,P_KHP,M_KHP,V_T1,V_T2,V_HCl,R]):
...     print( " {}: {:G}".format(i.label,i.u) )
...
R: 0.000101387
V_T2: 9.69953E-05
V_T1: 8.33653E-05
V_HCl: 7.39151E-05
m_KHP: 3.19376E-05
P_KHP: 2.9268E-05
M_KHP: 1.86931E-06
```

The results (which can be compared with Figure A3.6 in the Guide) show that repeatability is the dominant component of uncertainty

The full uncertainty budget is obtained by

```
>>> for i in rp.budget(c_HCl):
...    print( " {}: {:G}".format(i.label,i.u) )
...
 R: 0.000101387
 V_T2_cal: 8.33938E-05
 V_T1_cal: 6.66166E-05
 uV_HCl_cal: 5.51882E-05
 V_T1_temp: 5.01198E-05
 V_T2_temp: 4.95334E-05
 uV_HCl_temp: 4.91702E-05
 P_KHP: 2.9268E-05
 u_lin_tare: 2.25833E-05
 u_lin_gross: 2.25833E-05
 M_C: 1.83443E-06
```

This shows that calibration error in the volume titrated is also an important component of uncertainty .

Special aspects of this measurement

The CITAC Guide discusses some aspects of this measurement in section A3.6. Two in particular are: the uncertainty associated with repeatability and bias in titration volumes.

A reduction in the uncertainty attributed to repeatability, by a factor of $\sqrt{3}$, has a small effect on the final combined uncertainty. This may be seen in the GTC calculation by changing the definition of the uncertain number R

```
>>> R = ureal(1.0,0.001/math.sqrt(3),label='R')

>>> c_HCl = R * (1000 * m_KHP * P_KHP * V_T2)/(M_KHP * V_T1 * V_HCl)
>>> print('c_HCl ={}'.format(c_HCl))
c_HCl = 0.10139(16)


>>> for i in rp.budget(c_HCl,influences=[m_KHP,P_KHP,M_KHP,V_T1,V_T2,V_HCl,R]):
...     print( " {}: {:G}".format(i.label,i.u) )
...
 V_T2: 9.69953E-05
 V_T1: 8.33653E-05
 V_HCl: 7.39151E-05
 R: 5.85359E-05
 m_KHP: 3.19376E-05
 P_KHP: 2.9268E-05
 M_KHP: 1.86931E-06
```

The new results show that the combined uncertainty is not much changed when the repeatability is improved.

Another consideration is that a bias may be introduced by the use of phenolphthalein as an indicator. The excess volume in this case is about 0.05 mL with a standard uncertainty of 0.03 mL.

We can adapt our calculations above by defining two elementary uncertain numbers to represent the bias. These can be subtracted from the previous estimates[4]:

```
>>> V_T1_excess = ureal(0.05,0.03,label='V_T1_excess')
>>> V_T1 = V_T1 - V_T1_excess

>>> V_T2_excess = ureal(0.05,0.03,label='V_T2_excess')
>>> V_T2 = V_T2 - V_T2_excess

>>> print( uncertainty(V_T1) )
0.033688...

>>> print( uncertainty(V_T2) )
0.033210...
```

The uncertainties are roughly twice the previous values.

The concentration of HCl can then be re-calculated using the same measurement equation

```
>>> c_HCl = R * (1000 * m_KHP * P_KHP * V_T2)/(M_KHP * V_T1 * V_HCl)
>>> print( c_HCl )
0.10132(31)
>>> print( uncertainty(c_HCl) )
0.0003096...
```

[4] The CITAC Guide does not provide different raw titration results for this case. However, the numerical values of V_T1 and V_T2 will not be the same, because there are now two different parts to the experiment.

The combined uncertainty is now about twice as large (in mol/L).

Cadmium released from ceramic-ware (A5)

This section is based on a measurement described in Appendix 5 of the 3rd edition of the EURACHEM / CITAC Guide[1].

The CITAC Guide gives a careful discussion of the uncertainty analysis leading to particular numerical values. The following shows only how the data processing could be preformed using `GTC`.

The measurement

The experiment determines the amount of cadmium released from ceramic ware.

The measurand can be expressed as

$$r = \frac{c_0 \cdot V_\mathrm{L}}{a_\mathrm{v}} \cdot d \cdot f_\mathrm{acid} \cdot f_\mathrm{time} \cdot f_\mathrm{temp} \ ,$$

where

- r is the mass of cadmium leached per unit area $(\mathrm{mg\,dm^{-2}})$,
- c_0 cadmium content in the extraction solution $(\mathrm{mg\,L^{-1}})$,
- V_L is the volume of leachate (L),
- d is the dilution factor,
- a_v is the surface area of the liquid $(\mathrm{dm^2})$.
- f_acid is the influence of acid concentration.
- f_time is the influence of the duration,
- f_temp is the influence of temperature.

[1] On-line: http://www.citac.cc/QUAM2012_P1.pdf

The uncertainty contributions

Section A5.4 of the CITAC Guide provides numerical estimates of these quantities that can be used to define uncertain numbers for the calculation.

Dilution factor

In this example there was no dilution.

Leachate volume

Several factors contribute to the uncertainty of V_{L}:

- $V_{\mathrm{L-fill}}$ the relative accuracy with which the vessel can be filled
- $V_{\mathrm{L-temp}}$ temperature variation affects the determined volume
- $V_{\mathrm{L-read}}$ the accuracy with which the volume reading can be made
- $V_{\mathrm{L-cal}}$ the accuracy with which the manufacturer can calibrate a 500 mL vessel

Uncertain numbers for each contribution can be defined and combined to obtain an uncertain number for the volume. In this case, the volume of leachate is 332 mL.

```
>>> v_leachate = 332 # mL
>>> a_liquid = 2.1E-4   # liquid volume expansion per degree
>>> v_fill = ureal(0.995,tb.triangular(0.005),label='v_fill')
>>> v_temp = ureal(0,tb.uniform(v_leachate*a_liquid*2),label='v_temp')
>>> v_reading = ureal(1,tb.triangular(0.01),label='v_reading')
>>> v_cal = ureal(0,tb.triangular(2.5),label='v_cal')

# Change units to liters now
>>> V_L = result(
...     (v_leachate * v_fill * v_reading + v_temp + v_cal)/1000,
...     label='V_L') # L
...
>>> print( "V leachate: {}".format(V_L) )
V leachate: 0.3303(18)
```

A calibration curve for cadmium concentration

The amount of leached cadmium is calculated using a calibration curve. A linear relationship is assumed between observed absorbance and cadmium concentration.

$$A_i = c_i \cdot B_1 + B_0 + E_i \ ,$$

where B_1 and B_0 are the slope and intercept, respectively, of the line, A_i is the observed absorbance, c_i is the concentration of the i^{th} calibration standard and E_i is the unknown measurement error incurred during the i^{th} observation.

Three repeat observations are made for each of five calibration standards and the parameters of the calibration line are estimated by ordinary least-squares regression.

The GTC calculation uses the *line_fit()* function

```
>>> x_data = [0.1, 0.1, 0.1, 0.3, 0.3, 0.3, 0.5, 0.5, 0.5, 0.7, 0.7, 0.7, 0.9, 0.9, 0.
↪9]
>>> y_data = [0.028, 0.029, 0.029, 0.084, 0.083, 0.081, 0.135,
...             0.131, 0.133, 0.180, 0.181, 0.183, 0.215, 0.230, 0.216]
...
>>> fit = ta.line_fit(x_data,y_data,label='regression')

>>> B_0 = fit.intercept
>>> B_1 = fit.slope

>>> print( "Intercept: {}".format(B_0) )
Intercept: 0.0087(29)
>>> print( "Slope: {}".format(B_1) )
Slope: 0.2410(50)
```

There is correlation between these uncertain numbers (the estimates are correlated)

```
>>> print( get_correlation(B_0, B_1) )
-0.87038...
```

The object `fit` contains information about the regression that can be used to make predictions about cadmium concentration from subsequent observations of absorbance. In this case, two further values of absorbance are used to estimate the concentration c_0.

Using the function *x_from_y()* we write (the label 'absorbance' will be attached to the mean of the observations and identify this influence in the uncertainty budget below)

```
>>> c_0 = fit.x_from_y( [0.0712,0.0716], x_label='absorbance',y_label='noise' )
>>> print( "absorbance: {}".format(c_0) )
absorbance: 0.260(18)
```

Liquid surface area

There are two contributions to the uncertainty of a_{V}:

- $a_{\mathrm{V-dia}}$ uncertainty due to the diameter measurement
- $a_{\mathrm{V-shape}}$ uncertainty due to imperfect shape

Uncertain numbers for each contribution can be combined to obtain an estimate of the surface area

```
>>> dia = ureal(2.70,0.01,label='dia')
>>> a_dia = math.pi*(dia/2)**2
>>> a_shape = ureal(1,0.05/1.96,label='a_shape')
>>> a_V = result( a_dia * a_shape, label='a_V' )
>>> print( "a_V: {}".format(a_V) )
a_V: 5.73(15)
```

Temperature effect

The temperature factor is given as $f_{\text{temp}} = 1 \pm 0.1$. Assuming a uniform distribution we define

```
>>> f_temp = ureal(1,tb.uniform(0.1),label='f_temp')
```

Time effect

The time factor is given as $f_{\text{time}} = 1 \pm 0.0015$. Assuming a uniform distribution we define

```
>>> f_time = ureal(1,tb.uniform(0.0015),label='f_time')
```

Acid concentration

The acid concentration factor is given as $f_{\text{acid}} = 1 \pm 0.0008$. This is already in the form of a standard uncertainty so we define

```
>>> f_acid = ureal(1,0.0008,label='f_acid')
```

The uncertainty calculation

To estimate r we now evaluate

```
>>> r = c_0 * V_L / a_V * f_acid * f_time * f_temp
>>> print( "r: {}".format(r) )
r: 0.0150(14)
```

The contribution from the different influences can be examined

```
>>> for i in rp.budget(r,influences=[c_0,V_L,a_V,f_acid,f_time,f_temp]):
...     print( " {}: {:G}".format(i.label,i.u) )
...
 absorbance: 0.00102956
 f_temp: 0.00086663
 a_V: 0.000398736
 V_L: 8.28714E-05
 f_time: 1.29994E-05
 f_acid: 1.20084E-05
```

The results (which can be compared with Figure A5.8 in the Guide) show that the content of cadmium in the extraction solution is the dominant component of uncertainty.

The full uncertainty budget can be obtained by writing

```
>>> for i in rp.budget(r,trim=0):
...     print( " {}: {:G}".format(i.label,i.u) )
...
 noise: 0.000928623
 f_temp: 0.00086663
 a_regression: 0.000688685
 a_shape: 0.00038292
 b_regression: 0.000311899
 dia: 0.000111189
 v_reading: 6.128E-05
```

(continues on next page)

(continued from previous page)

```
v_cal: 4.63764E-05
v_fill: 3.0794E-05
f_time: 1.29994E-05
f_acid: 1.20084E-05
v_temp: 3.65814E-06
```

This reveals that the additional observations of absorbance have contributed most to the uncertainty (so perhaps a few more observations would help)

3.1.4 Arrays of uncertain numbers

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 in `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)),
```

(continues on next page)

(continued from previous page)

```
...  (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

FOUR

OTHER TOPICS

4.1 Some comments about GTC fitting functions

4.1.1 Overview

GTC has straight-line regression functions in both the *type_a* and *type_b* modules.

Functions in *type_a* implement a variety of least-squares straight line fitting algorithms that provide results in the form of uncertain numbers. When used, the input data (the sequences `x` and `y`) are treated as pure numbers (if sequences of uncertain numbers are provided, only the values are used in calculations).

Functions defined in *type_b*, on the other hand, expect input sequences of uncertain numbers. These functions estimate the slope and intercept of a line by applying the same type of regression algorithms, but uncertainties are propagated through the equations and the residuals are ignored.

The distinction between functions that evaluate the uncertainty of estimates from residuals (*type_a*) and functions that evaluate uncertainty using uncertain numbers (*type_b*) is useful. There will be circumstances that require the use of a function in *type_b*, such as when systematic errors contribute to uncertainty but cannot be estimated properly using conventional regression. Without the methods available in *type_b*, such components of uncertainty could not be propagated. On the other hand, functions in *type_a* implement conventional regression methods.

Discretion will be needed if it is believed that variability in a sample of data is due, in part, to errors not fully accounted for in an uncertain-number description of the data. The question is then: just how much of that variability can be explained by components of uncertainty already defined as uncertain number influences? If the answer is 'very little' then it will be appropriate to use a function from *type_a* to estimate the additional contribution to uncertainty from the sample variability. At the same time, components of uncertainty associated with the uncertain-number data should be propagated using a function from *type_b* that performs the same type of regression. The two result values will be identical (the estimates of the slope and intercept will be the same) but the uncertainties will differ. *type_a.merge()* can then be used to merge the results.

Clearly, this approach could potentially over-estimate the effect of some influences and inflate the combined uncertainty of results. It is a matter of judgement as to whether to merge type-A and type-B results in a particular procedure.

4.1.2 The type_a module regression functions

Ordinary least-squares

type_a.line_fit() implements a conventional ordinary least-squares straight-line regression. The residuals are used to estimate the underlying variance of the *y* data. The resulting uncertain numbers for the slope and intercept have finite degrees of freedom and are correlated.

Weighted least-squares

type_a.line_fit_wls() implements a so-called weighted least-squares straight-line regression. This assumes that the uncertainties provided with input data are known exactly (i.e., with infinite degrees of freedom). The uncertainties in the slope and intercept are calculated without considering the residuals.

This approach to linear regression is described in two well-known references[1][2] , but it may not be what many statisticians associate with the term 'weighted least-squares'.

Relative weighted least-squares

type_a.line_fit_rwls() implements a form of weighted least-squares straight-line regression that we refer to here as 'relative weighted least-squares'. (Statisticians may regard this as conventional weighted least-squares.)

type_a.line_fit_rwls() accepts a sequence of scale factors associated with the observations y, which are used as weighting factors. For an observation y, it is assumed that the uncertainty $u(y) = \sigma s_y$, where σ is an unknown factor common to all the y data and s_y is the weight factor provided.

The procedure estimates σ from the residuals, so the uncertain numbers returned for the slope and intercept have finite degrees of freedom.

Note, because the scale factors describe the relative weighting of different observations, the ordinary least-squares function *type_a.line_fit()* and *type_a.line_fit_rwls()* would return equivalent results if all y observations are given the same weighting.

Weighted total least-squares

type_a.line_fit_wtls() implements a form of least-squares straight-line regression that takes account of errors in both the x and y data[3].

As in the case of *type_a.line_fit_wls()*, the uncertainties provided for the x and y data are assumed exact. When calculating the uncertainty in the slope and intercept, the residuals are ignored and the uncertain numbers returned have infinite degrees of freedom.

[1] Philip Bevington and D. Keith Robinson, *Data Reduction and Error Analysis for the Physical Sciences*
[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical Recipes: The Art of Scientific Computing*
[3] M Krystek and M Anton, Meas. Sci. Technol. 22 (2011) 035101 (9pp)

4.1.3 The type_b module regression functions

Ordinary least-squares

type_b.line_fit() implements the conventional ordinary least-squares straight-line regression to obtain estimates of the slope and intercept of a line through the data. The y data is a sequence of uncertain numbers. The uncertainty of the slope and intercept is found by propagating uncertainty from the input data; the residuals are ignored.

Weighted least-squares

type_b.line_fit_wls() implements a weighted least-squares straight-line regression to estimate the slope and intercept of a line through the data. The y data is a sequence of uncertain numbers. An explicit sequence of uncertainties for the data points may also be provided. If so, these uncertainties are used as weights in the algorithm when estimating the slope and intercept. Otherwise, the uncertainty of each uncertain number for y is used. In either case, uncertainty in the estimates of slope and intercept is obtained by propagating the uncertainty associated with the input data through the estimate equations (the residuals are ignored).

Note: *type_a.line_fit_wls()* and *type_b.line_fit_wls()* yield the same results when a sequence of elementary uncertain numbers is defined for y and used with *type_a.line_fit_wls()* and the values and uncertainties of that sequence are used with *type_a.line_fit_wls()*.

Note: There is no need for a 'relative weighted least-squares' function in the *type_b* module. Using a sequence of u_y values with *type_b.line_fit_wls()* will perform this calculation.

Weighted total least-squares

type_b.line_fit_wtls() implements a form of least-squares straight-line regression that takes account of errors in both the x and y data.[Page 102, 3].

As with *type_b.line_fit_wls()*, sequences of uncertainties for the x and y data may be supplied in addition to sequences of the x and y data. When the optional uncertainty sequences are provided, estimates of the slope and intercept use those uncertainties as weights in the regression process. Otherwise, the input data uncertainties are used as weights in the regression process. In either case, uncertainty in the estimates of slope and intercept is calculated by propagating uncertainty from the input data through the regression equations (residuals are ignored).

CHAPTER

FIVE

RELEASE NOTES

5.1 License

```
MIT License

Copyright (c) 2022 Measurement Standards Laboratory of New Zealand

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

5.2 Developers

- Blair Hall
- Joseph Borbely

5.3 Release Notes

5.3.1 Version 1.3.8 (2022-04-01)

- A bug in function *reporting.sensitivity()* has been corrected. The function now returns zero when the independent term (`x`) in the argument has a standard uncertainty of zero. Previously, for example, an exception would be raised if the argument `x` was an uncertain complex number with a real or imaginary component having zero standard uncertainty. The docstring has been updated to explain the function's behaviour in more detail.
- Some improvements to the validation of arguments in the regression functions in *type_a* and *type_b*.

5.3.2 Version 1.3.7 (2022-02-24)

- The property `uid` has been added to *lib.UncertainReal* and *lib.UncertainComplex*. For an *lib.UncertainReal*, this returns the unique identifier of an elementary uncertain number and, when defined, the unique identifier associated with an intermediate uncertain number (otherwise it returns `None`). Analogous behaviour is defined for an *lib.UncertainComplex*.
- The namedtuple *named_tuples.Influence* has an attribute *uid* for the unique identifier of an uncertain-number influence. The function *reporting.budget()* returns a sequence of *Influence* objects, each of which now has a unique identifier. This is intended to permit indexing of arbitrary information about uncertain numbers held in other data structures.
- The idiom shown for using *reporting.budget()* has changed in the documentation. Previously, the pair of *Influence* attributes was expanded. Now, a temporary variable is assigned to each *Influence* item during iteration and then the required attributes are selected.
- A new section in the documentation under *reporting* has been added to provide an example that uses the new `uid` feature.
- The documentation in *straight_line_functions.rst* has been updated and some details corrected.

5.3.3 Version 1.3.6 (2021-09-06)

- When loading an uncertain-number archive from a file or string, if there is an existing intermediate `Node` registered with the context that is indistinguishable from another in the archive, then the existing `Node` will be used and no error is raised. `Node` objects are considered indistinguishable if their labels, uncertainties and degrees of freedom are the same.
- The string format of an uncertain real number now begins with a space, if the value is positive, or a negative sign.
- The string format of an uncertain complex number now always shows the sign (+ or -) of the real component

5.3.4 Version 1.3.5 (2021-08-19)

- Fixed a bug in *core.get_covariance()*. The function incorrectly returned zero when the arguments were a single independent elementary uncertain real number.
- Fixed a bug in *core.get_correlation()*. The function incorrectly returned zero when the arguments were a single independent elementary uncertain real number.
- *core.result()* now raises a warning if applied to a pure number and there is a label specified.
- The content of archive files generated by *persistence.dump()*, *persistence.dumps()*, *persistence.dump_json()* and *persistence.dumps_json()* has changed. Backward compatibility will be maintained until release 1.4.0.

5.3.5 Version 1.3.4 (2021-05-14)

- *reporting.budget()* now expects explicit keyword arguments for all options, instead of positional arguments (the names of the previous positional arguments are now the keywords).
- *reporting.budget()* takes a new key word `intermediate`
- *function.implicit()* has been added. This finds the uncertain-number solution `x` to a user-defined function $fn(x, ...) = 0$.

5.3.6 Version 1.3.3 (2021-02-16)

- Fixed an issue with merging uncertain numbers. The function *type_a.merge()* now has a tolerance parameter, which is used to determine whether the arguments a and b have equivalent values.

5.3.7 Version 1.3.2 (2020-09-16)

- Fixed an issue with restoration of archived uncertain numbers. A *RuntimeError* arose if two uncertain numbers, originally created in the same context, were restored to different archive objects in a new common context.
- An attempt to create a file or string representation of an empty archive raises a *RuntimeError*
- Docstrings for *add()* and *extract()* now mention the option of using the name as a look-up key (like a mapping)

5.3.8 Version 1.3.1 (2020-08-21)

- Fixed an issue with the *r* attribute of uncertain complex numbers, which returns the correlation coefficient between real and imaginary components: the calculation was incorrect (however, *core.get_correlation()* gave the correct result).
- Fixed an issue with the calculation of the variance-covariance matrix for an uncertain complex number with finite degrees of freedom: the matrix element for the variance of the real component was sometimes incorrectly returned for the variance of the imaginary component as well.

5.3.9 Version 1.3.0 (2020-07-28)

- Added support to *persistence* for archive storage in a JSON format. The new functions are: *persistence.dump_json()*, *persistence.dumps_json()*, *persistence.load_json()* and *persistence.loads_json()*

5.3.10 Version 1.2.1 (2020-04-01)

- Fixed issue #18 - calculate the inverse of a matrix with uncertain elements
- Revised the documentation for the *persistence* module

5.3.11 Version 1.2.0 (2019-10-16)

- Functions to perform straight-line regressions are included in modules *type_a* and *type_b*.
- The regression functions in *type_a* act on sequences of numerical data in the conventional sense (i.e., only the values of data are used; if the data include uncertain number objects, the associated uncertainty is ignored). The residuals are evaluated and may contribute to the uncertainty of the results obtained, depending on the regression method.
- The regression functions in *type_b* act on sequences of uncertain-numbers, propagating uncertainty into the results obtained. In most cases, the regression functions in this module are paired with a function of the same name in *type_a*. For example, *type_a.line_fit()* and *type_b.line_fit()* both perform an ordinary least-squares regression. The uncertain-numbers for the intercept and slope obtained from *type_a.line_fit()* are correlated and have uncertainties that depend on the fitting residuals. On the other hand, the intercept and slope obtained by *type_b.line_fit()* depend on the uncertain-number data supplied, and does not take account of the residuals.
- The function *type_a.merge()* may be used to combine results obtained from type-A and type-B regressions performed on the same data.

- A number of example calculations are included from Appendix H of the *Guide to the expression of uncertainty in measurement* (GUM).
- A number of example calculations are included from the 3rd Edition (2012) of the EURACHEM/CITAC Guide: *Quantifying Uncertainty in Analytical Measurement* (CG4).
- There are several examples of applying GTC to linear calibration problems, including the use of regression functions in *type_a* and *type_b*.

5.3.12 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).

5.3.13 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.

5.4 Indices and tables

- genindex
- modindex

PYTHON MODULE INDEX

INDEX

K

L

M

N

O

P

R

S

T

U

V

X

Y

Z