Automating metrological data processing

Blair Hall

Measurement Standards Laboratory of New Zealand

blair.hall@measurement.govt.nz

Digital Transformation in Metrology Workshop:
DCC for developer and its implementation in NQI

(21-23 August, 2023, Bangkok, Thailand).

Overview

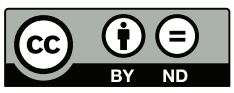
Data processing in metrology is often laborious, because calculations must include uncertainty analysis, and must be independently checked to meet the requirements of quality systems.

This tutorial introduces an approach to data processing that automatically performs the steps prescribed in the *Guide to the Expression of Uncertainty in Measurement* (GUM) for uncertainty calculations. This makes the development, validation, and execution of data processing much easier for metrologists.

The approach is a good example of digitalisation explicitly for metrology. A new data type, called an *uncertain number*, is able to hide the complexity of uncertainty analysis and meet metrological requirements to represent a traceability chain. The approach simplifies the development of data processing and enables new applications and services to be envisaged.

A copy of this document and the associated Python scripts has been archived at

http://doi.org/10.5281/zenodo.8245143



This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit

http://creativecommons.org/licenses/by-nd/4.0/

or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

This license allows for redistribution, commercial and non-commercial, as long as the work is passed along unchanged and in whole, with appropriate credit.

Preparation

Some familiarity with software development and object-oriented programming is assumed.

Code is written in Python 3, using a package called the GUM Tree Calculator (GTC). Documentation for GTC is available online (https://gtc.readthedocs.io). Instructions for installing the software are available:

https://gtc.readthedocs.io/en/stable/install.html.

An understanding of formal measurement modelling is assumed as well as familiarity with basic notions of measurement uncertainty. If modelling is not familiar, it is **strongly** recommended to read *An Introduction to Measurement Uncertainty*, by Hall and White, before the tutorial. The booklet is available:

https://doi.org/10.5281/zenodo.3872590.

Notation

Code is shown in fixed-width font with syntax highlighting, like `print('Hello')`, or in blocks on a coloured background, like

```
for i in range(10):
    print(i)
```

The notation introduced in *An Introduction to Measurement Uncertainty* is used [IMU]. Upper case letters denote quantities that are not known exactly, while lower case letters denote quantities that can be assigned numeric values. Such known values are usually estimates of quantities subject to uncertainty. For example, if Y is a measurand (which can never be exactly determined) then its estimate y is obtained by measurement.

Residual errors are denoted by the letter 'E' with a descriptive subscript.

For example, the relationship between a measurand, its estimate, and the residual error associated with the estimate could be expressed as

$$Y = y - E_y \ .$$

Estimates of residual errors are always zero, but they contribute to uncertainty and may sometimes lead to correlation between other estimates.

Note

The notation described in the slide is adequate for this tutorial.

However, alternatives may be needed to avoid conflict with conventional choices of symbols for particular quantities. In that case, unknown quantities may be underlined, like $\underline{y}$, and unknown quantities (estimates) decorated with a hat, like $\hat{y}$.

This notation may also be mixed with the upper and lower case style.

For example, the relationship between a measurand, $\underline{y}$, its estimate, $\hat{y}$, and the residual error associated with the estimate, $\underline{E}_{\hat{y}}$, could be either expressed as

$$\underline{y} = \hat{y} - \underline{E}_{\hat{y}}$$

or as

$$\underline{y} = y - E_y \ ,$$

using a mixture of notations.

Modelling

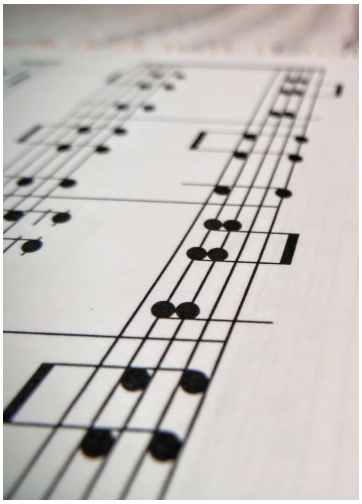
Part 1:

Modelling
- special notations
- the general model
- knowns and unknowns
- abstraction
- elementary uncertain number

Abstract notation

Many different notations are adopted in specialised fields

- Architectural drawings
- Chemical compounds
- Ishikawa (fish-bone) diagrams
- Electrical circuit diagrams
- Mathematical notations
- *Etc., etc.*



But what about metrology? We have measurement models expressed with quantity equations.

Quantity equations describe relations among *physical quantities* ($F = m\,a$, $R = V/I$).

Quantity equations do not take numeric values directly.

The simplest measurement model

We can apply the following equation to any measurement result

$$Y = y - E_y \ .$$

Y is the measurand and y the measured value. No measurement ever yields a perfect result. So, Y and y are *never* equal.

E_y is the measurement error.

Terms in capital letters are unknown quantities; we will not have numeric values for them; but the measurement result y will be known (the number obtained by measurement).

In practice we take y as an estimate of Y.

And we report a result with an uncertainty

$$y \ , \ u(y) \ ,$$

to provide information about likely values of E_y.

Notes

It is important to see that reporting an uncertainty with each result is a reflection of the most basic measurement concept: no measurement result is without error. So, there are two terms in our model and we report two numbers (although we can't report the error).

If we knew the value of E_y then the result y could be corrected to obtain an exact value for Y.

Metrologists study the behaviour of a measurement to accurately describe the dispersion of measurement errors.

In the GUM, the resulting distribution of measurement error is assumed to be a Gaussian with zero mean (unbiased). The combined *standard uncertainty* is an estimate of the standard deviation of this distribution.

Things you know that you don't know

When given numeric arguments, we can evaluate functions, but what if we do not know the actual values?

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

Each term consists of an estimate and a residual error

$$X_1 = x_1 - E_{x_1} \ , \textit{etc.}$$

All residual estimates are zero. So,

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

The unknown measurement error, E_y, results from the combined effects of dispersion in the x_i, which is due to the actual, but unknown, values of the E_i.

There are known knowns, things we know that we know; and there are known unknowns, things that we know we don't know. But there are also unknown unknowns, things we do not know we don't know.

- D. Rumsfeld

Notes

The Law of Propagation of Uncertainty (LPU), in the GUM, evaluates the sensitivity of Y to each input

$$c_i = \left. \frac{\partial Y}{\partial X_i} \right|_{X_i = x_i} \ .$$

However, because $X_i = x_i - E_{x_i}$ and the x_i are constant, these partial derivatives are essentially functions of the unknown residual errors (x_i values determine the point at which the derivative is evaluated)

$$c_i = -\left. \frac{\partial Y}{\partial E_{x_i}} \right|_{X_i = x_i} \ .$$

So, the calculation of a measured value and a GUM calculation of uncertainty use largely, if not completely, different sets of information about the measurement.

It is usually better to think of the general form of a measurement model as

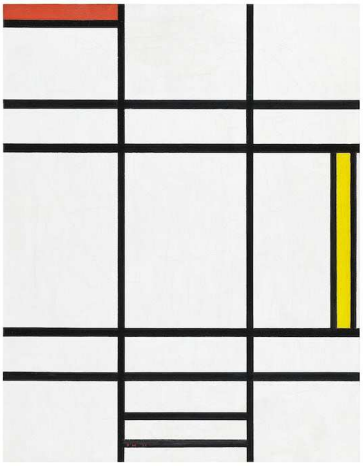
$$Y = f(E_1, E_2, \cdots ; x_1, x_2, \cdots) \ ,$$

where $x_1, x_2, \cdots$ are known values that determine the result, and $E_1, E_2, \cdots$ are residuals with zero estimates. The GUM expects all available corrections to be applied, which implies that estimates of residual errors are zero.

Abstraction

Matrices and complex numbers are mathematical abstractions and may use special notation.

Abstractions hide distracting, predictable, details in order to highlight more general concepts. For example, matrix notation hides a lot of computational steps (e.g., in matrix multiplication or inversion).



The *uncertain number* is an abstraction for metrology.

Uncertain numbers represent quantities. They encapsulate the estimate and other attributes related to uncertainty.

Mathematical operations apply to uncertain numbers.

Uncertain numbers hide the detailed steps of uncertainty calculations.

Software implementations of uncertain numbers can be used for data processing algorithms.

Notes

The uncertain number concept is described in a 2006 paper published in Metrologia called *Computing with uncertain numbers* [CUN].

Terminology: elementary uncertain numbers

We distinguish between inputs to a measurement model and quantities derived from those inputs.

The $X_1, X_2, \cdots$ are inputs to

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

The attributes of uncertain numbers representing inputs are assigned numbers (estimates, standard uncertainties, etc.).

We call these **elementary** uncertain numbers.

Other uncertain numbers are defined by expressions involving more uncertain numbers (elementary or not) or plain numbers.

We use the qualifier *elementary* only when it is important to know that we are referring to an input.

The GUM Tree Calculator

Part 2:

The GUM Tree Calculator

- data types
- `ureal()`
- unique identifiers
- systematic effects
- uncertainty budget
- sensitivity coefficients
- distribution types
- result
- classes and functions

The GUM Tree Calculator

GTC implements uncertain-number data types for real-valued and complex-valued quantities. This tutorial deals only with real-valued uncertain numbers.

The real-valued uncertain number attributes are:

- a value (the estimate)
- an uncertainty (standard uncertainty)
- a number of degrees of freedom

The data type provides additional attributes:

- uncertainty components
- a label (character string)
- a unique identifier (globally unique)

This section presents some of the features of the uncertain-number type.

The function `ureal()`

1. Start Python and import GTC
2. Create two uncertain numbers with `ureal()`
3. Use different values
4. Use an uncertainty of 1
5. Explore ...

```
>>> from GTC import *
>>> X = ureal(10,1,label='X')
>>> Y = ureal(5,1,label='Y')

>>> X
ureal(10.0,1.0,inf, label='X')

>>> print(X)
 10.0(1.0)

>>> value(X)
10.0

>>> uncertainty(X)
1.0

>>> label(X)
'X'

>>> X + Y
ureal(15.0,1.4142135623730951,inf)

>>> x / y
ureal(2.0,0.447213595499958,inf)
```

Notes

The slide shows an interactive Python session at the command line interface (e.g., the Windows console or Powershell Terminal)

The functions `value()`, `uncertainty()`, etc., have equivalent object attributes: `obj.x`, for value, `obj.u`, for uncertainty, `obj.label`, for label, etc. See the GTC documentation for details (https://gtc.readthedocs.io)

By default, the degrees of freedom is infinite. The Python `math.inf` representation of infinity is used.

Object identity and unique identifiers

Measurement Standards Laboratory of New Zealand

Uncertain-number objects represent quantity instances.

Quantities are unique (in time and space), so uncertain numbers should have this property too.

- Python variables refer to objects
- Python names are not bound to objects
- Python objects are only uniquely identified during a session

GTC assigns a globally unique identifier to every uncertain-number object.

Every object gets a different identifier—the same identifier will never be used twice, ever!

```
>>> Z = X
>>> Z
ureal(10.0,1.0,inf, label='X')

>>> Y, X = X, Y    # swap
    references

>>> X
ureal(5.0,1.0,inf, label='Y')
>>> Y
ureal(10.0,1.0,inf, label='X')

>>> id(X)
1957033608864

>>> id(Y)
1957033045792

>>> uid(X)
(27422642433611..., 2)
>>> uid(Y)
(27422642433611..., 1)
```

Notes

Python variable names just refer to objects in memory; these names can be reassigned.

When arithmetic operations are performed, it is the objects referred to by Python names that are involved.

GTC generates identifiers by combining a UUID (Universally Unique Identifier), which is a 128-bit number created at the beginning of each session, with the integer value of a counter of uncertain number objects.

By giving a unique identifier to each object, we can save and reuse the results of uncertain number calculations. This means we can pass on results obtained in one session to another, later, time and place. This is exactly what needs to be done to support metrological traceability.

Systematic effects: why identity matters

Many measurements are influenced by quantities that do not change from one measurement to the next.

These are systematic effects.

The magnitude of the effect may be reduced by taking differences or ratios.

GTC handles this automatically.

`E_s` represents a systematic residual error.

When we take the difference between two observations, the systematic effect disappears.

When we take the sum, it increases.

```
>>> E_s = ureal(0,1,label='E_s')

>>> O_1 = E_s + X
>>> print(Obs_1)
 10.0(1.4)

>>> O_2 = E_s + Y
>>> print(Obs_2)
 5.0(1.4)

>>> O_dif = O_1 - O_2
>>> print(O_dif)
 5.0(1.4)

>>> O_sum = O_1 + O_2
>>> print(O_sum)
 15.0(2.4)
```

Notes

We could write this in terms of measurement models:

$$O_1 = E_\mathrm{s} + X\,, \quad \text{and}$$
$$O_2 = E_\mathrm{s} + Y\,.$$

The sum and difference are

$$O_1 + O_2 = X + 2E_\mathrm{s} + Y\,, \quad \text{and}$$
$$O_1 - O_2 = X - Y\,.$$

According to the GUM,

$$u(o_1 + o_2) = \sqrt{u^2(x) + 4 \cdot u^2(e_\mathrm{s}) + u^2(y)}$$

and

$$u(o_1 - o_2) = \sqrt{u^2(x) + u^2(y)}\,.$$

However, if you were to ignore the systematic effect, and treat the two observations as independent, the sum and difference would have the same combined uncertainty. That is,

$$u(o_1 \pm o_2) = \sqrt{u^2(o_1) + u^2(o_2)}\,.$$

An uncertainty budget for the influences

GTC can enumerate the individual contributions from terms that influence an uncertain number.

This is an uncertainty budget.

We can also see that `O_dif` is influenced by `X` and `Y` by evaluating the correlation between them.

On the other hand, `O_dif` is not correlated to `E_s`, which means `E_s` has no influence on `O_dif`.

```
>>> for x_i in rp.budget(O_sum, trim=0):
...     print( f'{x_i.label}: {x_i.u}' )
...
E_s: 2.0
X: 1.0
Y: 1.0

>>> for x_i in rp.budget(O_dif, trim=0):
...     print( f'{x_i.label}: {x_i.u}' )
...
X: 1.0
Y: 1.0
E_s: 0.0

>>> get_correlation(O_dif,X)
0.7071067811865475

>>> get_correlation(O_dif,Y)
-0.7071067811865475

>>> get_correlation(O_dif,E_s)
0.0
```

Notes

There are a number of keyword argument options for `rp.budget()` (see reporting.budget). The keyword `trim` controls the reporting of components with little or no effect on the uncertainty. By setting `trim = 0`, we get all uncertainty components.

Uncertainty components and sensitivities

To make it easier to think about our results, we have set all elementary uncertain number uncertainties to unity.

This means the components of uncertainty are also the partial derivatives.

`component()` returns the absolute value.

`u_component()` returns the signed uncertainty component.

`sensitivity()` returns the signed partial derivative.

```
>>> component(O_dif,X)
1.0
>>> component(O_dif,Y)
1.0
>>>

>>> O_dif.u_component(X)
1.0
>>> O_dif.u_component(Y)
-1.0

>>> O_dif.sensitivity(X)
1.0
>>> O_dif.sensitivity(Y)
-1.0
```

Notes

The GUM defines a component of uncertainty as the product of the magnitude of a partial derivative with a standard uncertainty

$$u_i(y) = \left| \frac{\partial Y}{\partial X_i} \right| u(x_i) \ ,$$

which is always positive.

Signed components of uncertainty,

$$u_i(y) = \frac{\partial Y}{\partial X_i} u(x_i) \ ,$$

are important, so GTC provides access to these too.

Input distribution types

The GUM uses different types of probability distribution to describe the unpredictable effects of influence quantities.

The uniform, triangular, and arcsine distributions are standard alternatives to the Gaussian distribution. These distributions are more often characterised by their half-width than by their standard deviation.

GTC functions convert a half-width into a standard deviation, which is required by `ureal()`.

```
>>> tb.uniform(1)
0.5773502691896258

>>> tb.triangular(1)
0.4082482904638631

>>> tb.arcsine(1)
0.7071067811865475

>>> ureal(0, tb.arcsine(1))
ureal(0.0,0.7071067811865475,inf)
```

Notes

The theory behind the GUM method for evaluating uncertainty is concerned with variances and covariances (correlations). It applies to any types of distribution (any mixture) provided the measurement model is linear.

However, the GUM method of calculating an uncertainty interval, with a particular level of confidence, from the standard uncertainty assumes that the distribution of measurement error is approximated by a Gaussian. This is very often a satisfactory approximation in metrology.

The uncertain-number `result()` function

Intermediate results in a calculation can be declared by `result()`.

Once declared, we may evaluate a component of uncertainty, or a sensitivity coefficient, with respect to an intermediate result.

`result()` can label an uncertain number.

`result()` must be used if uncertain numbers need to be stored.

```
>>> I = ureal(1.3E-3,0.01E-3)
>>> R = ureal(995,7)
>>> V = I*R
>>> P = V**2/R
>>> component(P,V)
Traceback (most recent call last):
(...)
RuntimeError: ‘x‘ is not an
     elementary or intermediate
     uncertain number

>>> V = result( I*R, label='V' )
>>> P = V**2/R
>>> component(P,V)
3.505784505642068e-05

>>> P.sensitivity(V)
0.0026
```

User-defined classes and functions

Measurement Standards Laboratory of New Zealand

Classes are useful for data processing.

Objects maintain their own state. For example, each instance (object) of this DVM class can have a different serial number.

Python functions and methods (functions bound to a class) can take uncertain-number arguments.

The function `average()` works for any sequence of objects that can be added together.

```
>>> class DVM(object):
...     def __init__(self,sn):
...         self.serial_number = sn
...
>>> dvm1 = DVM("Acme #1")
>>> dvm2 = DVM("Acme #2")

>>> print( dvm1.serial_number )
Acme #1
>>> print( dvm2.serial_number )
Acme #2

>>> def average( lst ):
...     total = sum(lst)
...     return total / len(lst)
...
>>> data = [
...     ureal(1,1), ureal(2,1),
     ureal(3,1)
... ]
>>> average(data)
ureal(2.0,0.5773502691896257,inf)
```

Notes

Arguments to a Python function can be of any type. An error is raised if the objects do not support the operations applied to them.

The function `average()` takes a sequence of objects (numbers or uncertain numbers). For `sum()` to work, these objects must have a defined behaviour for addition to each other, and for `average()` to work, a defined behaviour for division by a number (`len()` returns an integer).

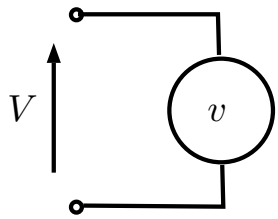
Examples

Part 3:

Examples
- simple voltmeter
- calibration
- type-A and type-B
- linear calibration

Simple voltmeter model

Here we consider a simple measurement model of a voltmeter.



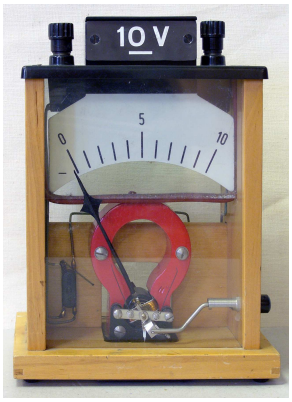
The reading v_i is the response to the actual applied voltage

$$V_i = (1 + E_{\text{rel}} + E_{\text{rel}\cdot i})\, v_i - E_{\text{off}} - E_{\text{off}\cdot i}$$

$E_{\text{off}\cdot i}$ and $E_{\text{rel}\cdot i}$: independent random errors (noise) in the i^{th} reading

E_{off} : a fixed offset that shifts each reading by the same amount (a zero setting error)

E_{rel} : a fixed scale-factor error (gain error).



Hannes Grobe, *Schulhistorische Sammlung*, CC BY 3.0
https://commons.wikimedia.org/w/index.php?curid=4626470

Notes

We obtain the model by considering the various influences on the stimulus.

We can build up an equation for what is read from the meter in steps:

$$\begin{aligned} v_i &\approx V_i \\ &\approx V_i + E_{\text{off}\cdot i} \\ &\approx V_i + E_{\text{off}} + E_{\text{off}\cdot i} \\ &\approx \frac{V_i + E_{\text{off}} + E_{\text{off}\cdot i}}{1 + E_{\text{rel}}} \\ &= \frac{V_i + E_{\text{off}} + E_{\text{off}\cdot i}}{1 + E_{\text{rel}} + E_{\text{rel}\cdot i}} \end{aligned}$$

The equals sign on the last equation is intended to show that the model is complete. That is, we consider it sufficiently detailed for our requirements (whatever they may be). It follows that we could determine V_i without significant error if we knew the actual values of E_{off}, $E_{\text{off}\cdot i}$, $E_{\text{rel}\cdot i}$ and E_{rel}. (Of course, we do not know E_{off}, $E_{\text{off}\cdot i}$, $E_{\text{rel}\cdot i}$ and E_{rel}.)

This equation for v_i is called an observation equation. It can be rearranged to make V_i the subject, as shown on the slide (which is the *measurement model*).

Provided measurements are made on a particular DVM scale, a simple model like this may be adequate for many applications.

Implementing the model

A voltmeter model class.

Fixed influences (offset and gain errors) are object attributes.

An uncertain number is created each time `measure()` is called (random noise).

Voltmeter indications (numbers) can be transformed into uncertain numbers by `measure()`.

```
class VM(object):

    def __init__(self,E_off,E_rel,u_off_i,u_rel_i):
        self.E_off = E_off
        self.E_rel = E_rel
        self.u_off_i = u_off_i
        self.u_rel_i = u_rel_i
        self.count = 0

    def measure(self,v):
        self.count += 1
        lbl = f'E_off_{self.count}'
        E_off_i = ureal(0,self.u_off_i,label=lbl)
        lbl = f'E_rel_{self.count}'
        E_rel_i = ureal(0,self.u_rel_i,label=lbl)
        tmp = 1 + self.E_rel + E_rel_i

        return v * tmp - self.E_off  - E_off_i
#-----------------------------------------------
E_rel = ureal(0,8E-4,label='E_rel')
E_off = ureal(0,5E-5,label='E_off')
vm = VM(E_off=E_off,E_rel=E_rel,
        u_off_i=8E-5,u_rel_i=5E-5)
```

Notes

The code shown is in the file `vm.py`.

To execute this code, open a command prompt in the folder containing example scripts and type:

```
python -i vm.py.
```

The option `-i` tells the Python interpreter to remain active after executing the script and wait for more input to be typed in on the command line (on some systems, the interpreter may be invoked by just the two letters `py`).

To exit the interactive mode, type CTRL-Z followed by RETURN. However, don't do that yet.

Use the uncertain-number voltmeter

`vm.measure()` turns raw readings into applied-voltage estimates (uncertain numbers).

The scale-factor error dominates the uncertainty budget.

The number in the label for random errors increments each time `measure()` is called.

The influence of fixed effects reduces in the voltage difference.

The uncertainty in the voltage difference would be 0.0059 (≈ 8 times greater) if GTC did not account for common influence factors.

```
>>> V1 = vm.measure(5.1)
>>> show(V1)
ureal(5.1,0.004119201985822011,inf)
    E_rel: 0.00408
    E_off: 0.0005
    E_rel_1: 0.000255
    E_off_1: 8e-05
>>> V2 = vm.measure(5.2)
>>> show(V2)
ureal(5.2,0.004198761722222399,inf)
    E_rel: 0.0041600000000000005
    E_off: 0.0005
    E_rel_2: 0.00026000000000000003
    E_off_2: 8e-05
>>> show(V2 - V1)
ureal(0.100...,0.0003896...,inf)
    E_rel_2: 0.00026000000000000003
    E_rel_1: 0.000255
    E_rel: 8.000000000000021e-05
    E_off_1: 8e-05
    E_off_2: 8e-05
    E_off: 0.0
```

Notes

This slide uses a function named `show()` to display an uncertain number and its components of uncertainty. The function is defined as

```
def show(x):
    """
    Print the uncertain number and its uncertainty budget

    """
    if x.label is None:
        print(f'{x!r}')
    else:
        print(f'{x.label} = \t{x.x}, {x.u}, {x.df}')
    for u_i in rp.budget(x,trim=0):
        print(f'    {u_i.label}: {u_i.u}')
```

Work with collections of data

`vmmeasure()` can also be used with a sequence of readings.

A list of uncertain numbers is produced from 5 readings.

We can operate on this sequence as if it contained plain numbers (most standard mathematical operations are also defined for uncertain numbers).

The results are uncertain numbers.

```
>>> raw = [ 3.1, 4.2, 5.3, 5.9, 7.1 ]
>>> V = [ vm.measure(i) for i in raw ]
>>> V
[ureal(3.1,0.002535...,inf), ureal
     (4.2,0.003404...,inf), ureal
     (5.3,0.004278...,inf), ureal
     (5.9,0.004756...,inf), ureal
     (7.1,0.005713...,inf)]

>>> show( sum(V)/len(V) )
ureal(5.12,0.004128...,inf)
    E_rel: 0.004096
    E_off: 0.0005
    E_rel_7: 7.1e-05
    E_rel_6: 5.900...e-05
    E_rel_5: 5.3e-05
    E_rel_4: 4.200...e-05
    E_rel_3: 3.1e-05
    E_off_3: 1.600...e-05
    E_off_4: 1.600...e-05
    E_off_5: 1.600...e-05
    E_off_6: 1.600...e-05
    E_off_7: 1.600...e-05
```

Notes

Remember a few slides ago we created the `vm` object with `u_off_i = 8E-5`. Here, the uncertainty budget for the average of 5 readings shows that each random error contributes just 1.6E-5 to the uncertainty. This is because the sum of the objects in the sequence has been divided by `len(V) = 5`, reducing the influence of each error by a factor of 5 .

On the other hand, the fixed offset error has exactly the same influence on the average as it does on each estimate.

The influence of the scale-factor error depends on the size of each reading (it is a proportional effect), but it is not reduced by averaging either.

We could do the maths

The model of one estimate is

$$V_i = (1 + E_{\text{rel}} + E_{\text{rel}\cdot i})v_i - E_{\text{off}} - E_{\text{off}\cdot i}$$

and the average of 5 estimates is

$$= \frac{1}{5}\sum_{i=1}^{5}[(1 + E_{\text{rel}} + E_{\text{rel}\cdot i})v_i - E_{\text{off}} - E_{\text{off}\cdot i}]$$

$$\approx \frac{1}{5}\sum_{i=1}^{5} v_i(1 + E_{\text{rel}} + E_{\text{rel}\cdot i}) - E_{\text{off}} - \frac{1}{5}\sum_{i=1}^{5} E_{\text{off}\cdot i}$$

Averaging reduces the influence of random noise, but not the fixed effects of offset and scale factor errors.

As problems become more complicated, analysis becomes laborious (albeit possible). Notation is often a difficulty. Here, we used the subscript i from the outset. However, our notation would not allow for more complicated scenarios.

Notes

A slightly more complicated problem is to take two sequences of voltage readings, average them, and then take the difference. Our notation would need an extra subscript to adequately represent this. Something like

$$V_{i\cdot j} = (1 + E_{\text{rel}} + E_{\text{rel}\cdot i\cdot j})v_{i\cdot j} - E_{\text{off}} - E_{\text{off}\cdot i\cdot j}\ ,$$

where $j = 1, 2$ would identify sequences 1 or 2, and i would index readings in a sequence.

GTC automatically handles models like this, as discussed earlier, without special attention. Each uncertain number object is uniquely identified, so it can never be confused with another uncertain number object.

Calibration

Now consider a system with measurable systematic effects, S_{off} and S_{rel}.

The response to some stimulus X_i is (the *observation equation*)

$$x_i = X_i(1 + S_{\text{rel}}) + S_{\text{off}} + E_{\text{rnd}\cdot i} \ ,$$

where $E_{\text{rnd}\cdot i}$ is an independent random error in each reading.

Calibrated reference standards R_{a} and R_{b} can be used to measure S_{off} and S_{rel}.

- Measure the response to each standard (x_{a} and x_{b})
- Let $Y_{\text{a}} = x_{\text{a}} - E_{\text{rnd}\cdot\text{a}}$, and similarly $Y_{\text{b}} = x_{\text{b}} - E_{\text{rnd}\cdot\text{b}}$.
- Solve for $1 + S^{\text{c}}_{\text{rel}}$ and $S^{\text{c}}_{\text{off}}$.

$$\begin{bmatrix} R_{\text{a}} & 1 \\ R_{\text{b}} & 1 \end{bmatrix} \begin{bmatrix} 1 + S^{\text{c}}_{\text{rel}} \\ S^{\text{c}}_{\text{off}} \end{bmatrix} = \begin{bmatrix} Y_{\text{a}} \\ Y_{\text{b}} \end{bmatrix} \ .$$

The solution is influenced by random errors ($Y_{\text{a}} \approx x_{\text{a}}$ and $Y_{\text{b}} \approx x_{\text{b}}$) and the calibration standards.

Notes

When the standards are measured, the system responses (observation equations) are

$$\begin{aligned} x_{\text{a}} &= R_{\text{a}}(1 + S_{\text{rel}}) + S_{\text{off}} + E_{\text{rnd}\cdot\text{a}} \\ x_{\text{b}} &= R_{\text{b}}(1 + S_{\text{rel}}) + S_{\text{off}} + E_{\text{rnd}\cdot\text{b}} \ , \end{aligned}$$

with random unbiased (expectation of zero) errors for each measurement.

We define $Y_{\text{a}} = x_{\text{a}} - E_{\text{rnd}\cdot\text{a}}$ and $Y_{\text{b}} = x_{\text{b}} - E_{\text{rnd}\cdot\text{b}}$, so that

$$\begin{aligned} Y_{\text{a}} &= R_{\text{a}}(1 + S_{\text{rel}}) + S_{\text{off}} \\ Y_{\text{b}} &= R_{\text{b}}(1 + S_{\text{rel}}) + S_{\text{off}} \ , \end{aligned}$$

which can be expressed as a matrix product.

However, Y_{a}, Y_{b}, R_{a}, and R_{b} are all subject to uncertainty. We would solve the system of equations using the best estimates of these terms and obtain approximate values for the systematic effects, which are denoted $S^{\text{c}}_{\text{rel}}$ and $S^{\text{c}}_{\text{off}}$.

Correction

Correction may enhance the accuracy of the system.

Each raw reading x_i could be corrected for the systematic effects

$$x_i' = \frac{x_i - S^{\mathrm{c}}_{\mathrm{off}}}{1 + S^{\mathrm{c}}_{\mathrm{rel}}} \ .$$

The result x_i' is likely to be a better estimate of the stimulus X_i than the raw response x_i.

However, $S^{\mathrm{c}}_{\mathrm{rel}}$ and $S^{\mathrm{c}}_{\mathrm{off}}$ depend on the values of the references used to calibrate the system and on the random system noise during calibration, all of which are uncertain.

It is not easy to express the uncertainty of x_i' as an estimate of X_i.

Notes

The values of $S^{\mathrm{c}}_{\mathrm{rel}}$ and $S^{\mathrm{c}}_{\mathrm{off}}$ depend on errors in the calibrated values of R_{a} and R_{b} and on the random errors $E_{\mathrm{rnd\cdot a}}$ and $E_{\mathrm{rnd\cdot b}}$. The solution of the linear equation does not explicitly show the form of this dependence.

Implementation

A class can define a method that incorporates raw-reading correction.

When `MeasurementSystem` objects are created, the default values of the correction terms have no effect but an uncertain number will be generated to represent random system noise.

Correction values can be set using `calibrate()`.

```
class MeasurementSystem(object):

    def __init__(self,S_off_c=0,S_rel_c=0,
     u_rnd=0.01):
        self.S_off_c = S_off_c
        self.S_rel_c = S_rel_c
        self.u_rnd = u_rnd
        self.count = 0

    def measure(self,x,label=None):
        if label is None:
            self.count += 1
            label = f'E_rnd_{self.count}'

        E_rnd = ureal(0.0,self.u_rnd,label=
     label)

        num = x - self.S_off_c - E_rnd
        den = 1 + self.S_rel_c
        return num / den

    def calibrate(self,S_off_cal,S_rel_cal):
        self.S_off_c = S_off_c
        self.S_rel_c = S_rel_c
```

Correction terms

Correction terms can be stored for later reuse (assuming the measuring system is stable).

After restoring these terms, corrected uncertain numbers can be generated from raw data.

These uncertain numbers are influenced by errors arising during the calibration of the system and the reference standards.

The various influences are retained by the uncertain-number structure. So, traceability could be established back to primary standards.

```
with open('cal_coefficients.json','r') as f:
    a = pr.load_json(f)

    S_rel_c = a.extract('S_rel_c')
    S_off_c = a.extract('S_off_c')

ms = MeasurementSystem(
    S_off_c=S_off_c,
    S_rel_c=S_rel_c
)
```

```
>>> X = ms.measure(.5)
>>> show(X)
ureal(0.513413...,0.009775...,inf)
    E_rnd_1: 0.009725943624440295
    E_rb: 0.000512939232645745
    E_rnd_x_b: 0.0004988818059476182
    E_ra: 0.0004870607673542549
    E_rnd_x_a: 0.0004737125564964114
```

Using reference standards

Calibrated reference standard values are restored from a file.

We would normally measure the system response to reference standards. For the tutorial, we simulate that step using a `VirtualMeasurementSystem`.

The uncertain-number responses generated by `ms.measure()` correspond to Y_a and Y_b.

The calibration equations were shown earlier.

```
with open('stds.json','r') as f:
    a = pr.load_json(f)

    R_a = a.extract('R_a')
    R_b = a.extract('R_b')
#-------------------------------------------
vms = VirtualMeasurementSystem(
    S_off=-0.02,S_rel=0.03)

x_a = vms.measure(R_a)
x_b = vms.measure(R_b)
#-------------------------------------------
ms = MeasurementSystem( u_rnd=1E-2 )

Y_a = ms.measure(value(x_a),label=f'E_rnd_x_a'
    )
Y_b = ms.measure(value(x_b),label=f'E_rnd_x_b'
    )
#-------------------------------------------
A = la.uarray([[R_a,1],[R_b,1]])
B = la.uarray([Y_a,Y_b])
S = la.solve(A,B)

S_rel_c= result( S[0]-1, label='S_rel_c' )
S_off_c= result( S[1], label='S_off_c' )
```

Notes

The `VirtualMeasurementSystem` class simulates a measurement system response to a stimulus. No uncertain numbers are involved. Objects are assigned values for static offset and scale-factor errors during construction. These values would be unknown in a real situation. However, they can be chosen for the purpose of simulations.

The observation equation is applied when `measure()` is called, applying values for the static offset and scale-factor errors to the given stimulus. A Gaussian random variate is added to simulate random system noise.

A floating-point number is returned by `measure()`.

The `VirtualMeasurementSystem` class is defined in `vms.py`. The measurement system calibration is handled in `cal.py`.

```
from random import gauss, seed

seed(123456789)

class VirtualMeasurementSystem(object):

    def __init__(self,S_off=0,S_rel=1,u_rnd=0.01):
        self.S_off = S_off
        self.S_rel = S_rel
        self.u_rnd = u_rnd

    def measure(self,X):
        """
        Return a simulated value for the response
        to a stimulus 'X'

        """
        # Add random noise to generate raw data
        E_rnd = gauss(0.0,self.u_rnd)
        tmp = value(X)*(1 + self.S_rel)

        return  tmp + self.S_off + E_rnd
```

Calibrated reference standards

Suppose the reference standards were calibrated in a different laboratory that provided conventional results. We take those values, and uncertainties, and create corresponding uncertain numbers.

`R_a` and `R_b` could have been created directly as elementary uncertain numbers. The more general idiom works for more than one error.

An `Archive` object collects the information needed to restore uncertain numbers in a different Python session.

```
r_a = 0.0015
u_ra = 1E-3

E_ra = ureal(0,u_ra,label='E_ra')

R_a = result(r_a - E_ra, label='R_a')

#-----------------------------------
r_b = 0.9995
u_rb = 1E-3

E_rb = ureal(0,u_rb,label='E_rb')

R_b = result(r_b - E_rb, label='R_b')

#-----------------------------------
a = pr.Archive()
a.add(R_a=R_a,R_b=R_b)

with open('stds.json','w') as f:
    pr.dump_json(f,a,indent=4)
```

Notes

We express the reference-standard quantities as

$$R_\mathrm{a} = r_\mathrm{a} - E_\mathrm{ra} \ ,$$
$$R_\mathrm{b} = r_\mathrm{b} - E_\mathrm{rb} \ ,$$

where the reported values are r_a and r_b and the residual errors E_ra and E_rb represent the combined calibration errors.

When an external (paper-based) calibration report is the only information available (i.e., without an uncertainty budget), this is how calibrated values can be handled (including storage). This represents the measurement results correctly, because the unknown residual errors are unique to the reported values.

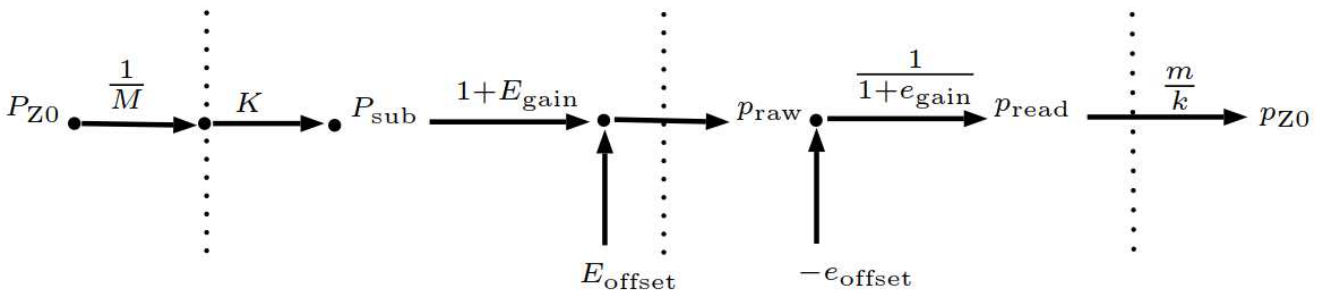
The code shown in the slide is in `cal_standards.py`.

To sum up

- Uncertain numbers form links along traceability chains.
- Uncertain-number results can be stored from one session, then reloaded and used in another.
- Very complicated measurements can be handled by processing data in stages. Uncertain-number calculations are never finished: another stage can always be added.
- Uncertain-number code closely follows the mathematical description of measurement models and observation equations.
- The challenge is to develop correct models—uncertainty calculations are now automatic.
- We must check that code implements the model equations and that the model equations adequately describe the measurement.

Notes

A similar but more complicated example of using uncertain numbers is microwave power measurement. The figure below shows a signal flow graph for a measurement. From the left, the measurand P_{Z0} flows to the right and, because of mismatch error and the sensor calibration factor, a power level P_{sub} is detected by the meter. Instrumentation errors in the gain and offset lead to internal value for this power p_{raw}. Instrument correction factors are applied, producing the displayed value p_{read}. This may be further corrected, using estimates of the mismatch error and calibration factor, to obtain the measured value p_{Z0}.



In the reference section, at the end of these notes, several other publications describe advanced applications of GTC.

Uncertainty evaluation: type-A and type-B

The GUM distinguishes between type-A evaluation of uncertainty (statistical analysis of a sample of data) and type-B (not using statistical analysis methods).

GTC has a module for type-A evaluation of uncertainty, which defines functions that act on values of objects—if they are uncertain numbers, the uncertainty is ignored.

Another module supports type-B uncertainty evaluation, with functions that act on uncertain numbers , rather than numbers.

Sometimes it is desirable to merge results from both methods—`ta.merge()` does this.

```
>>> x
[0.8266062796279688, 1.0972031172443724,
     0.9273915184503791,
     0.980193924994468,
     0.9513306802341795,
     0.7772608339175362,
     1.0376040971180425,
     1.0415515674558646,
     1.057473868490803,
     1.1635125760607454]
>>> ta.mean(x)
0.9860128463594359
>>> ta.estimate(x)
ureal(0.9860...,0.03776...,9)

>>> x_un = [
...     ureal(x_i,1,label=f'x_{i}')
...         for i,x_i in enumerate(x) ]

>>> ta.merge(
...     ta.estimate(x,label="type-A"),
...     tb.mean(x_un) )
ureal(0.9860...,0.318...,45500.15...)
```

Notes

Apply `show()` to the uncertain-number result to see the uncertainty contributions.

Note, `merge()` offers a useful defensive programming technique. If the parameters of the measurement model are set conservatively, including a type-A evaluation should not significantly alter the uncertainty: the type-A component should have little effect. However, this component will become large if any unexpected variability in data does arise, which is very helpful for quality assurance (I speak from personal experience!).

Unexpected variability may be due to some part of the equipment not operating as expected. (So, the source of unusual variability should be identified as soon as possible.)

The merged uncertain-number will have a type-A contribution describing the real dispersion of the data. So, the combined uncertainty will remain objective, albeit a little conservative.

```
>>> show ( ta.merge(
...     ta.estimate(x,label="type-A"),
...     tb.mean(x_un) ) )
ureal(0.9860...,0.3184...,45500.1...)
    x_0: 0.1
    x_1: 0.1
    x_2: 0.1
    x_3: 0.1
    x_4: 0.1
    x_5: 0.1
    x_6: 0.1
    x_7: 0.1
    x_8: 0.1
    x_9: 0.1
    type-A: 0.03776879128444202
```

(Code used is in the file `types_ab.py`.)

Linear calibration functions

Sometimes a measurement system is described by a linear model, like

$$Y_i = A + B\,X_i + E_i \ ,$$

where E_i is an unbiased independent random error.

The inverse function predicts the stimulus from a response

$$X_i = (Y_i - A - E_i)/B \ .$$

Processing the responses to a set of reference points (stimuli) $\{X_i\}$ may be used to measure A and B.

Data processing (with uncertain numbers) uses ordinary least-squares regression.

There are type-A and type-B regression functions in GTC.

Notes

The usual statistical assumption is that E_i is drawn from a population of Gaussian errors with no bias (a mean of zero) and an unknown variance, σ^2. In this tutorial we use ordinary least-squares (OLS). There are alternative regression algorithms in GTC, which are appropriate for different models.

Type-A OLS regression

The data $\{y_i\}$ are numerical values.

Uncertain-number results include degrees of freedom and the correlation between slope and intercept.

The uncertainty depends on dispersion in the sample.

```
x = [ 1., 2., 3., 4., 5., 6. ]
y = [ 3.014, 5.225, 7.004, 9.061, 11.201,
     12.762 ]
```

```
>>> result_A = ta.line_fit(x,y)
>>> print(result_A)

Ordinary Least-Squares Results:

  Intercept:  1.17(16)
  Slope:  1.964(41)
  Correlation: -0.9
  Sum of the squared residuals: 0.1164...
  Number of points: 6

>>> A, B = result_A.a_b
>>> A
ureal(1.1719...,0.1588...,4)
>>> B
ureal(1.9635...,0.04079...,4)
>>> get_correlation(A,B)
-0.8987170342729172
```

Notes

GTC type-A regression functions treat the $\{Y_i\}$ data as simple numerical values (the associated uncertainty is ignored if uncertain numbers are used). So, the type-A regression functions are similar to conventional data processing algorithms in other software packages. However, the GTC functions return uncertain-number results. Usually, the estimates of A and B are correlated.

The data for this example comes from Appendix E of the British Standard *Determination and use of straight-line calibration functions* (DD ISO/TS 28037:2010).

Our use of this data is essentially the same as in the Appendix but our processing is different. The Appendix initially considers all uncertainties to be unity and uses uncertainty propagation to obtain parameter uncertainties. On the next slide, we obtain the same result using a type-B OLS regression function. In a second step, the Appendix uses the sum of squared residuals (chi-squared) to obtain better estimates of the parameter uncertainties. We obtain type-A parameter estimates directly by OLS regression.

(The data are in `ols.py`.)

Type-B OLS regression

The $\{Y_i\}$ are uncertain numbers.

Uncertain-number results include correlation between slope and intercept but do not depend on the fitted residuals. The number of degrees of freedom is infinity.

The values found for slope and intercept are the same using type-A and type-B analyses.

However, the parameter uncertainties are different. Type-B uncertainties are propagated from the $\{Y_i\}$ uncertainties (all unity).

```
>>> Y = [  ureal(y_i,1) for y_i in y ]
>>> result_B = tb.line_fit(x,Y)
>>> print(result_b)

Ordinary Least-Squares Results:

  Intercept:  1.17(93)
  Slope:  1.96(24)
  Correlation: -0.9
  Sum of the squared residuals:
     0.11649...
  Number of points: 6

>>> A, B = result_B.a_b
>>> A
ureal(1.1719...,0.9309...,inf)
>>> B
ureal(1.9635...,0.2390...,inf)
>>> get_correlation(A,B)
-0.898717034272917
```

Using OLS results

The object returned as a result has information about the fit (see previous slides) and two methods:

`x_from_y()` predicts a stimulus X corresponding to a sequence of response values.

`y_from_x()` predicts a response Y to a stimulus X. An uncertain number can be used for X, in which case the associated uncertainty is propagated.

```
>>> result_A.x_from_y([5])
ureal(1.9495089...,0.09924...,4.0)

>>> result_A.x_from_y([5,5.1,4.95])
ureal(1.957996...,0.06930...,4.0)

>>> A, B = result_A.ab
>>> Y_new = ta.estimate([5,5.1,4.95] )
>>> (Y_new - A)/B
ureal(1.957996...,0.06930...,5.43121...)



>>> result_B.x_from_y([5,5.1,4.95])
ureal(1.957996...,0.2801...,inf)

>>> result_B.x_from_y( [ta.estimate
    ([5,5.1,4.95])] )
ureal(1.957996...,0.2801...,49041.9...)
```

Notes

When a sequence of data is provided to `x_from_y()` method for the type-A result, the inverse function is applied to the average value of the responses. The uncertainty of the stimulus obtained depends on the response sample standard deviation, but the size of the sample is not considered (the degrees of freedom remains the same as the parameter estimates).

However, the degrees of freedom is taken into account by an uncertain-number calculation corresponding to

$$X_{\text{new}} = \frac{Y_{\text{new}} - A}{B} \ ,$$

with uncertain numbers for the parameters A and B, and an uncertain number for Y_{new}.

Closing

Part 3:

Closing
- RF power application
- development process
- conclusion

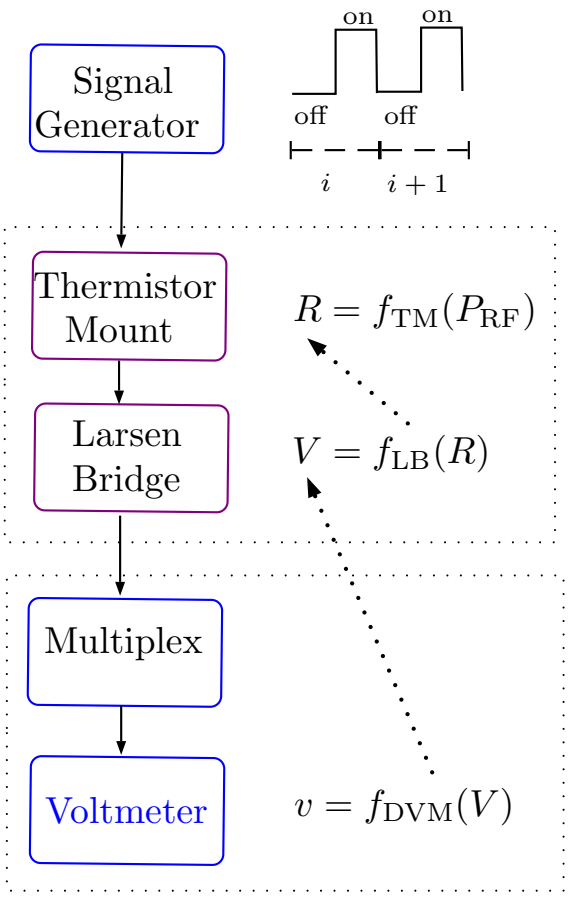
Notes

This last section discusses how the ideas presented so far can be used.

An example is given showing how the method is really helpful at breaking down complicated problems into smaller conceptual pieces. This is taken from an application of uncertain numbers to data processing in a radio and microwave frequency laboratory. Other examples are given in the references at the end of the notes.

A big-picture description of the proposed methodology is given. Many benefits are possible if this approach is followed. Measurement procedures can be documented more thoroughly and software can be developed and tested against clearer requirements based on scientific models.

RF Power



Power is measured by the change in resistance (between power on and power off) of a thermistor mount.

The bridge generates a voltage proportional to resistance.

The change in voltage is related to incident power by $P_{\mathrm{RF}} = (V_{\mathrm{on}}^2 - V_{\mathrm{off}}^2)/R_0$, where R_0 is a bridge parameter.

We average adjacent 'off' readings to reduce drift

$$V'_{\mathrm{off}\cdot i} = \frac{1}{2}(V_{\mathrm{off}\cdot i} + V_{\mathrm{off}\cdot i+1})$$

Raw readings ($v_{s\cdot i}$) are converted to uncertain numbers, using DVM and Larsen Bridge models, before being used in calculations.

Notes

The thermistor mount is a very stable reference standard used by NMIs. It is a passive device, which is connected to a special bridge to make RF power measurements (the Larsen Bridge, or Type-IV NBS Bridge, is one possibility). The bridge output is a voltage, which we measured, via a multiplexing switch, with a precision digital voltmeter.

The RF power must be switched on and off in a regular sequence. Measurements recorded raw voltage readings, which reflect the bridge balance conditions when power is on and off. The raw data are collected without using uncertain numbers. Data processing with uncertain numbers is applied to collections of raw data.

A measurement model of the calibrated DVM was used to convert raw readings into voltage estimates. A model of the Larsen Bridge and thermistor was used to covert voltage estimates to RF power estimates.

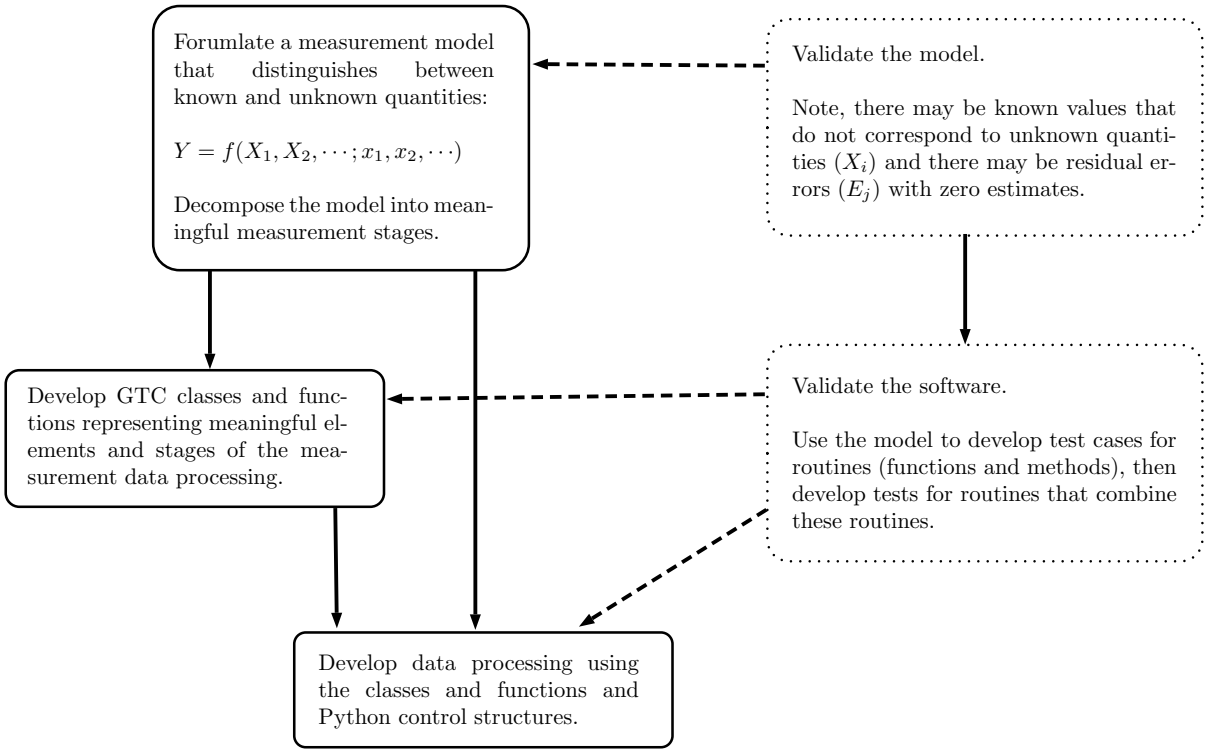
Each part of the measurement could be described with its own model. This simplified an otherwise complicated problem. It also allowed many different types of power sensor, bridges, and other kinds of power meter to be used.

To compensate for thermal drift, the average of adjacent 'off' voltage estimates is used. A series of power readings were averaged to provide a type-A characterisation of the level of remaining fluctuation in the data. The various systematic and random influences were accounted for by the uncertain-number algorithms.

Detailed uncertainty budgets were helpful in understanding the performance of the measurement system.

Two thermistor mount parameters are calibrated as a function of frequency (calibration factor and input reflection coefficient). The calibrated values were used to correct the power levels estimated by the bridge for a particular thermistor's response (using the calibration factor) and for network effects in the measurement set-up (input reflection coefficient).

How to approach development—the big picture



Notes

It is important to develop meaningful models of the measurement. Models can represent steps rather than a single 'whole-measurement' function. GTC automatically links the different stages together.

The scientific concepts related to a measurement should be explicit (measurement principles), the influences should be explicit (residual errors, etc.), the types of data that will be generated by measurement should be explicit (data management), and the processing steps that will be performed on data should be explicit. All of those things should be clearly documented.

Software can model real objects in the measurement domain, like sensors and instruments (see previous slide). These software objects can be incorporated in data processing. In this way the software closely resembles the description of the real world used in the model (a bit like a digital twin).

There will be a close relationship between data acquisition (e.g., a number of repeat measurements that a user could specify) and data processing routines. In the RF example of the previous slide, the same control structure and configuration parameters were used for data acquisition and data processing. Software objects were designed to switch between acquisition mode and processing mode.

Validation and testing is made easier by formal model equations that relate to measurement objects and processes. It is usually possible to test the different elements individually before putting together the overall data processing routines.

Note, only a small range of instrument capabilities is typically used in a measurement. For instance, only one particular range on a digital voltmeter may be used. The software model of the instrument need only cover the specific use. It is not necessary to develop a model covering all of the instrument functions!

Conclusion

- Uncertain numbers are a useful concept for metrology.
- Based on conventional scientific ideas and language (quantity equations and measurement errors) and designed to represent metrological traceability.
- An example of digitalisation for (the science of) metrology.
- Improve quality and enhance efficiency.
- A measurement model is linked to data processing, including rigorous, automated, calculation of measurement uncertainty.
- Less time required to measure and check results.
- Uncertain numbers can be serialised.

References

[GUM]: BIPM, IEC, IFCC, ILAC, ISO, IUPAC, IUPAP and OIML, *Evaluation of Measurement Data—Guide to the Expression of Uncertainty in Measurement (GUM 1995 with minor corrections)*, JCGM 100:2008 (Sèvres, France: Joint Committee for Guides in Metrology, 2008) http://www.bipm.org/en/publications/guides/gum.html

[IMU]: B. D. Hall and D. R. White, *An Introduction to Measurement Uncertainty*, (Measurement Standards Laboratory of New Zealand, 2020). https://doi.org/10.5281/zenodo.3872590

[CUN]: B. D. Hall, Computing with uncertain numbers, *Metrologia* **43**(6):L56 (2006). https://doi.org/10.1088/0026-1394/43/6/N07

[GTC]: B. D. Hall, The GUM Tree Calculator: A Python package for measurement modelling and evaluating measurement uncertainty, *Metrology* **2** 128–149 (2022). https://doi.org/10.3390/metrology2010009

[WIL]: Robin Willink, *Measurement Uncertainty and Probability*, (Cambridge University Press, 2013)

Advanced use of GTC

[MSD]: Ellie Molloy, *Metrology of Scattering Distributions*, (Ph.D. Thesis, Victoria University of Wellington, 2023). https://doi.org/10.26686/wgtn.23735001

[REG]: E. Molloy, P. Saunders and A. Koo, Effects of Rotation Errors on Goniometric Measurements, *Metrologia* (59) (2022). https://doi.org/10.1088/1681-7575/ac438e

[KCL]: B. D. Hall and A. Koo, Digital Representation of Measurement Uncertainty: A Case Study Linking an RMO Key Comparison with a CIPM Key Comparison, *Metrology* **1**(2) 166–181 (2021). https://doi.org/10.3390/metrology1020011

[VNA]: B. D. Hall, *Evaluating measurement uncertainty when calibration equations assume ideal standards*, Callaghan Innovation Report 592 (June, 2018). (Available from: https://www.researchgate.net/profile/Blair-Hall)