GUM Tree Calculator

Release 0.9.10

Measurement Standards Laboratory of New Zealand

Jul 22, 2016

Part I

License

CHAPTER

ONE

LICENSE

```
End User License Agreement
==========================

This license agreement (the "Agreement") is entered into between you, as a
private person or a company (the "Licensee") and Callaghan Innovation, a Crown
agency of the New Zealand government, having its registered address at
69 Gracefield Road, Lower Hutt, New Zealand ("CI"). By installing,
copying or otherwise using all or any portion of the GUM Tree Calculator
you ('the Licensee') agree to be bound by the terms of the Agreement.
If you do not agree to the terms of this Agreement do not install or
use the Software Product.

The Software Product is protected by Copyright and other intellectual property
laws and treaties. CI reserves all of its rights in the Software Product that
are not expressly granted under the terms of this Agreement.

1. LICENSE

Subject to the terms of this Agreement, CI hereby grants to Licensee a non-
exclusive, non-transferable, non-sublicensable and limited license only to
install and use one copy of the Software Product.

2. DEFINITIONS

"Software Product" means (a) the GUM Tree Calculator included third party
software files and other information and (b) upgrades, updates and additions to
such provided to you by CI, to the extent not provided under a separate
agreement.

3. LIMITATIONS AND RESTRICTIONS

The following shall apply in addition to the limitations and restrictions set
forth elsewhere in this Agreement:

Licensee shall not modify, adapt, translate or create derivative works based
upon the Software Product. Licensee shall not reverse engineer, decompile,
disassemble or otherwise attempt to discover the source code of the Software
Product except to the extent permitted by law.

Licensee shall not use, distribute, rent, lease, disclose or license out the
Software Product to its own end users or third parties.

Licensee shall not transfer any of its rights under this Agreement without the
express written consent of CI.

Licensee shall ensure all processing performed by the Software Product is
initiated by a local human user only. The Software Product may not be used as
part of a Web application or other server software.
```

```
4. SUPPORT AND MAINTENANCE

E-mail support: CI will attempt to respond to technical questions made by the
Licensee by email within a reasonable period of time.

Upgrades: CI will attempt to distribute upgrades to the Software Product on a
regular basis. Installation and use of such upgrades will be governed by the
terms of this Agreement.

5. THIRD PARTY RIGHTS

Any software provided along with the Software Product that is supplied under the
terms of a separate license agreement is subject to the terms of that license
agreement. This license does not apply to those portions of the Software
Product. Copies of those third party licenses are included in all copies of the
Software Product.

6. PRIVACY

The Licensee grants CI the right to store contact details and other related
information in order for CI to contact the Licensee occasionally regarding the
Software Product. CI will not use contact details and other related information
for any other purpose.

7. TERM AND TERMINATION

The Agreement and the licensing rights provided to the Licensee shall continue
until terminated. Without prejudice to any other rights, CI may terminate the
Agreement if the Licensee fails to comply with the terms and conditions of this
Agreement.

8. NO WARRANTY

TO THE MAXIMUM EXTENT PERMITTED BY LAW, CI DISCLAIMS ANY WARRANTY FOR
THE SOFTWARE. THE SOFTWARE, THE SERVICES AND ANY RELATED DOCUMENTATION
IS PROVIDED ON AN "AS IS" BASIS WITHOUT WARRANTY OF ANY KIND, WHETHER
EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR
NON-INFRINGEMENT OF THE RIGHTS OF THIRD PARTIES.

9. EXCLUSION

TO THE MAXIMUM EXTENT PERMITTED BY LAW, IN NO EVENT SHALL CI BE LIABLE
FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES
WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS
PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, PERSONAL
INJURY, LOSS OF PRIVACY OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE
USE OF OR INABILITY TO USE THE SOFTWARE EVEN IF ADVISED OF THE POSSIBILITY
OF SUCH DAMAGES.

10. GOVERNING LAW

This Agreement shall be governed by and construed in accordance with New Zealand
law and shall be subject to the non-exclusive jurisdiction of the  New Zealand
Courts.
```

Part II

Getting started

CHAPTER

TWO

INSTALLATION

2.1 Obtaining GTC

GTC is available without charge from http://mst.irl.cri.nz. Registration is required and use of the tool is subject to a non- exclusive, non-transferable and non-sublicensable limited end-user license agreement (*License*).

2.2 Upgrading

Before upgrading GTC on the computer, the previous version must be removed (see *Uninstalling*). Then follow the installation instructions (see *Installing*).

Note:

- Prior to version 0.9.9, GTC was installed under Program files, or Program files (x86), in the Windows file system (typically, C:\Program files (x86)\GTC).
- If you have one of these earlier versions, remove GTC (see *Uninstalling*) and copy any user-defined files to a safe place, then delete the old GTC installation folder completely.
- After installing the new version of GTC, copy any user-defined files to an appropriate folder and update the GTC_LIB environment variable if necessary (see *Windows environment variables*).
- By default, the installation creates a user lib folder under My GTC in the user's home directory (typically, e.g., C:\Users\user.name). If user-defined modules are copied here, there is no need to update the GTC_LIB environment variable.

2.3 Installing

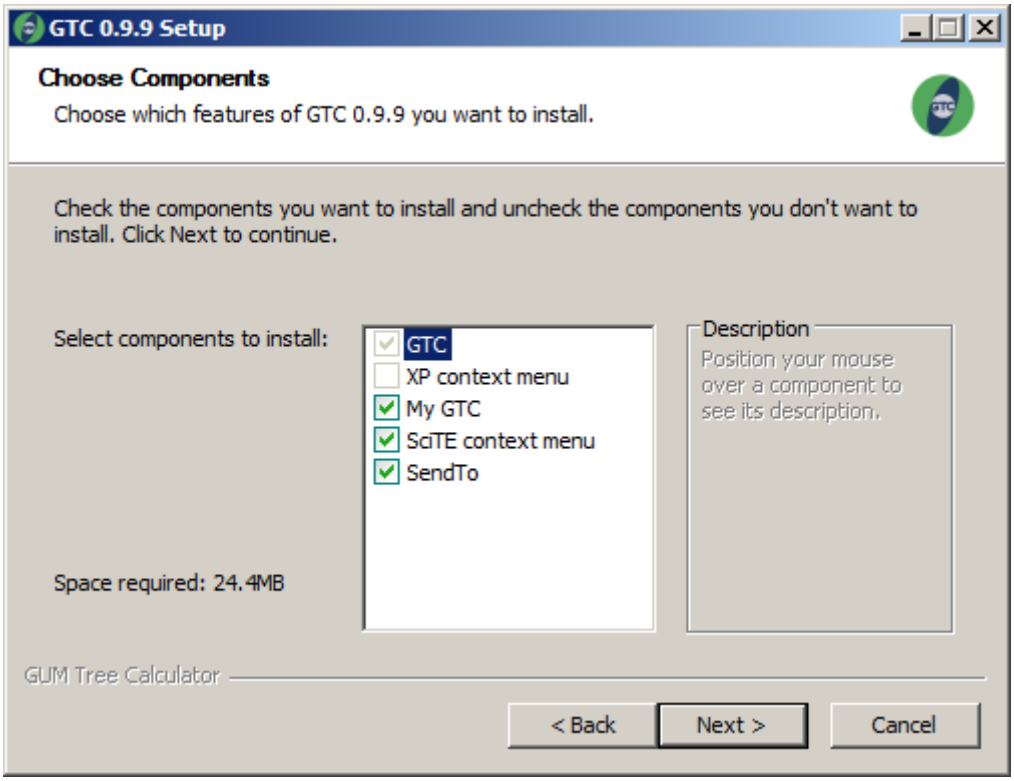
Run setup.exe to install the GUM Tree Calculator software.

Note:

- Installation is for a single user only (administrator rights are not required during installation).
- The software will be installed under the user's `AppData` folder (for example, `C:\Users\users.name\AppData\Local\GTC`)
- A user's `AppData` folder is hidden in the Windows Explorer by default (this can be changed in the Windows Explorer 'Folder options' settings).
- The user's `path` environment variable is modified and options are added to the Windows Explorer *Extended Context Menu* (see also *Windows environment variables*).
- A `GTC` entry is created in the Windows Start Menu. This contains shortcuts to the `GTC` command prompt (see *The Command Prompt*), as well as help files and an editor (see *The SciTE editor*).

Several options in the installer software relate to the extended Windows Explorer context menu.

- The context menu items for the *SciTE* editor and *SendTo* for `GTC` can be installed.
- A user `My GTC` folder can be created, in which case a subfolder `examples` will be created, containg some of the example files referred to in this manual, and a `lib` folder will be created and the path added to the user's `GTC_LIB` environment variable (see *Windows environment variables*) and a `scripts` folder will be created and added to the user's `GTC_SCRIPTS` environment variable.
- On Windows XP machines, an item that allows you to open a command window can be installed (in more recent Windows versions, this is always there by default).



2.4 Uninstalling

The *Add / Remove Programs* utility of the *Windows Control Panel* can be used to remove the software.

Alternatively, there is a link to `uninstall.exe` in the `GTC` Start Menu folder. The `uninstall.exe` program is located in the installation folder (e.g., `C:\Users\users.name\AppData\Local\GTC`).

Note:

- Uninstalling `GTC` does not remove user-defined modules that have been stored under the installation folder (e.g., in `C:\Users\users.name\AppData\Local\GTC\lib`), nor does it affect any user-made changes to the environment variable `GTC_LIB`.

2.5 Documentation

Several forms of documentation are provided, which can all be accessed from shortcuts in the `GTC` group in the Windows Start menu.

An on-line version of documentation is also available: http://www.uncertainnumbers.com/gtc/manual.

CHAPTER

THREE

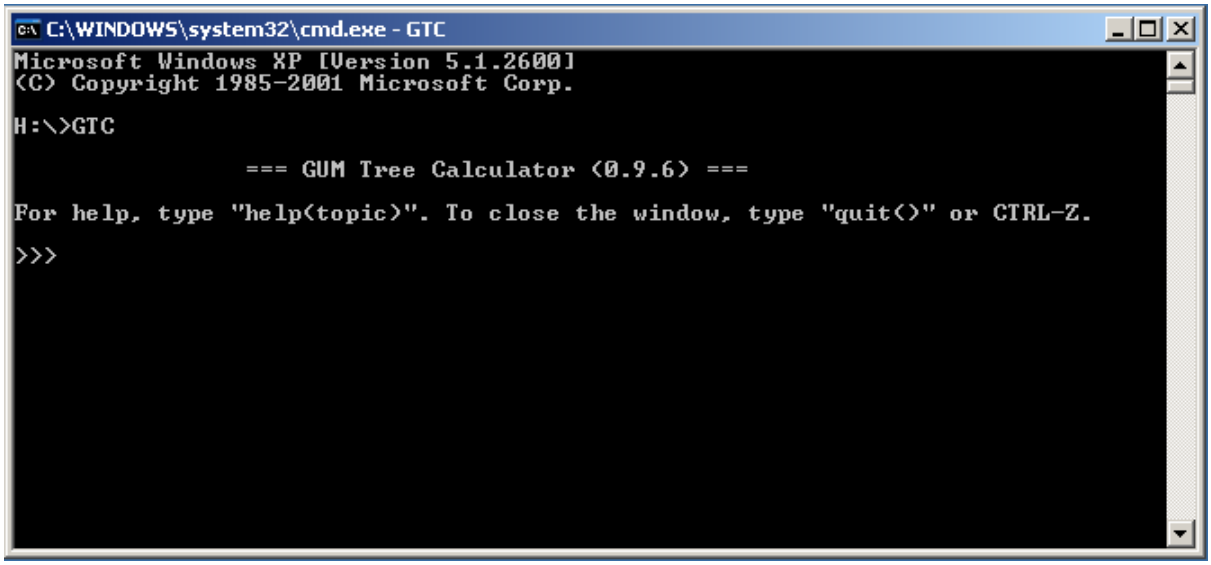
OPERATION

The GUM Tree Calculator (`GTC.exe`) can be activated from the Windows Command Prompt, or from the Windows Explorer extended context menu.

The SciTE editor, with context sensitive syntax highlighting and help facilities, is also provided.

3.1 The Command Prompt

There are several ways to use `GTC` from the Windows command prompt (see also *Windows command prompt syntax*).



1. When run without arguments, `GTC` enters an interactive mode.

   For instance:

```
>>> print copyright
Copyright (c) 2014, Callaghan Innovation. All rights reserved.
>>> print version
```

```
0.9.9
>>> 2**16 + 1
65537
```

To close the command window, type:

```
quit()
```

or `CTRL-Z` (press the `CTRL` key and the `Z` key together) at the command prompt.

2. `GTC` can execute data-processing scripts passed to it on the command-line

   For example, a file `hello.py` that contains:

```
msg = 'Hello'
print(msg)
```

   can be executed by:

```
C:\> gtc hello.py
Hello
```

   A series of scripts can be passed to `GTC` on a single command line, allowing calculations to be chained

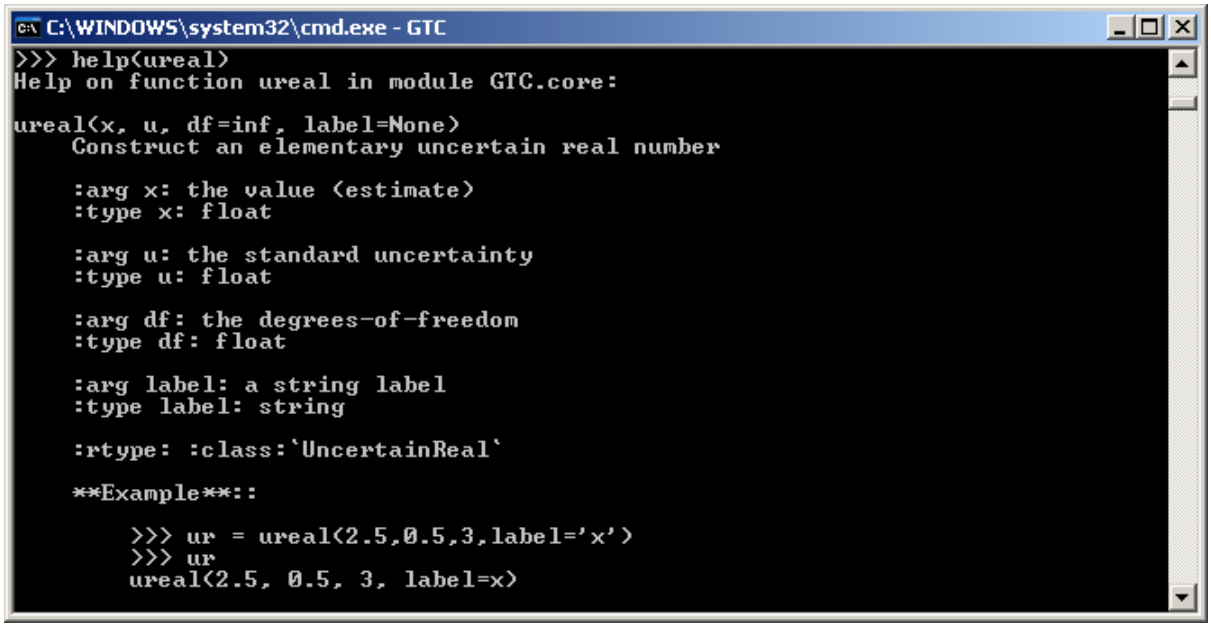
3. After processing scripts, `GTC` can be placed in an interactive mode, allowing further commands to be entered in by hand. The command-line switch `-i` selects interactive mode, e.g.:

```
C:\> gtc -i hello.py
Hello

>>> print(msg)
Hello
>>>
```

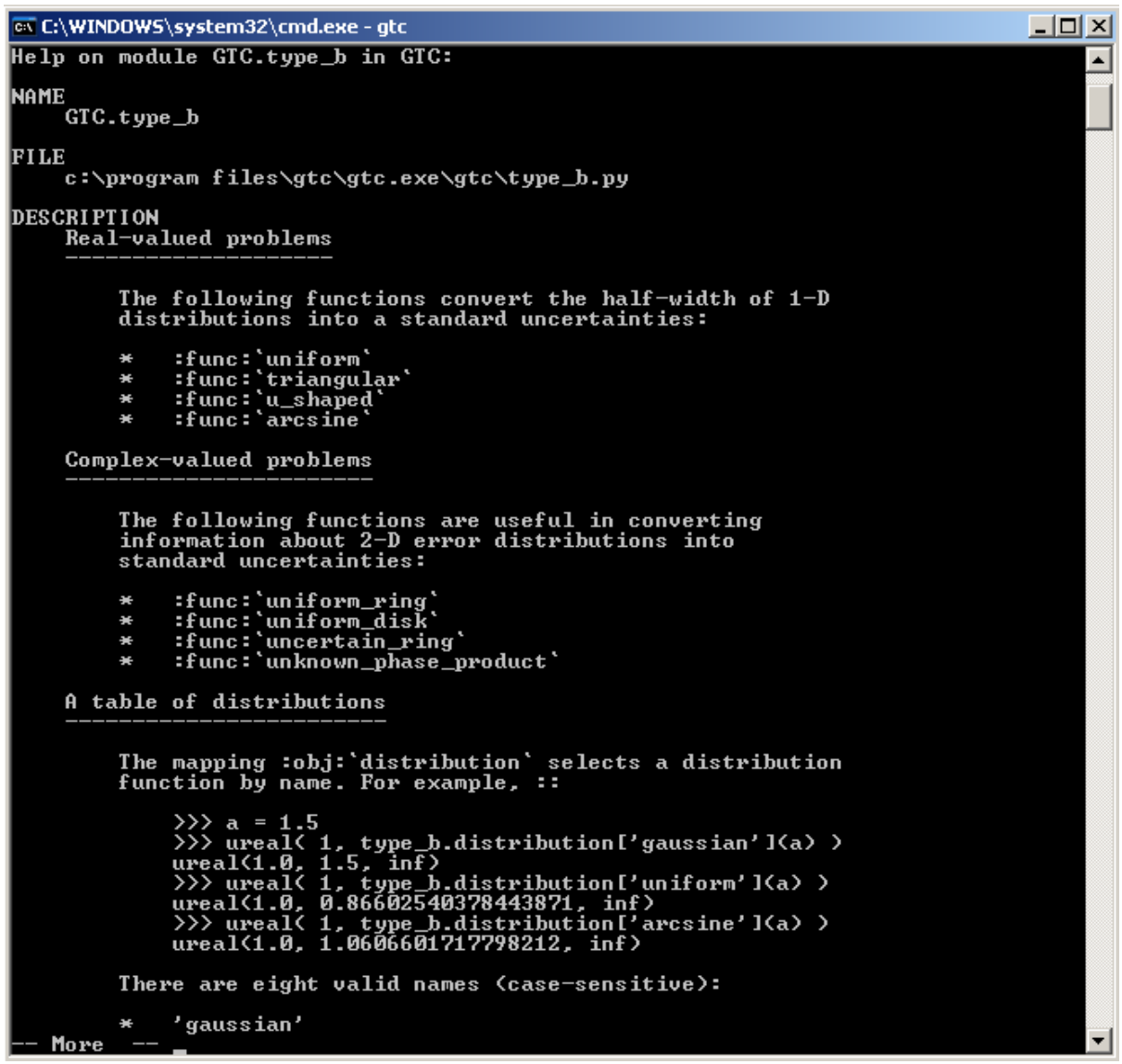
Help

Information about a particular `GTC` command, or module, can be obtained by typing `help(subject)` at the command prompt [1].

[1] When Help is displayed in the Command Prompt, some embedded formatting (mark-up) is visible. For example, `:arg:`, `:type:` and `:rtype:` are formatting commands.



If the help text is too long to fit in the command window, `-- more --` will appear at the bottom. To display the next line of text, press `ENTER`, or to display the next full page of text press the space bar.



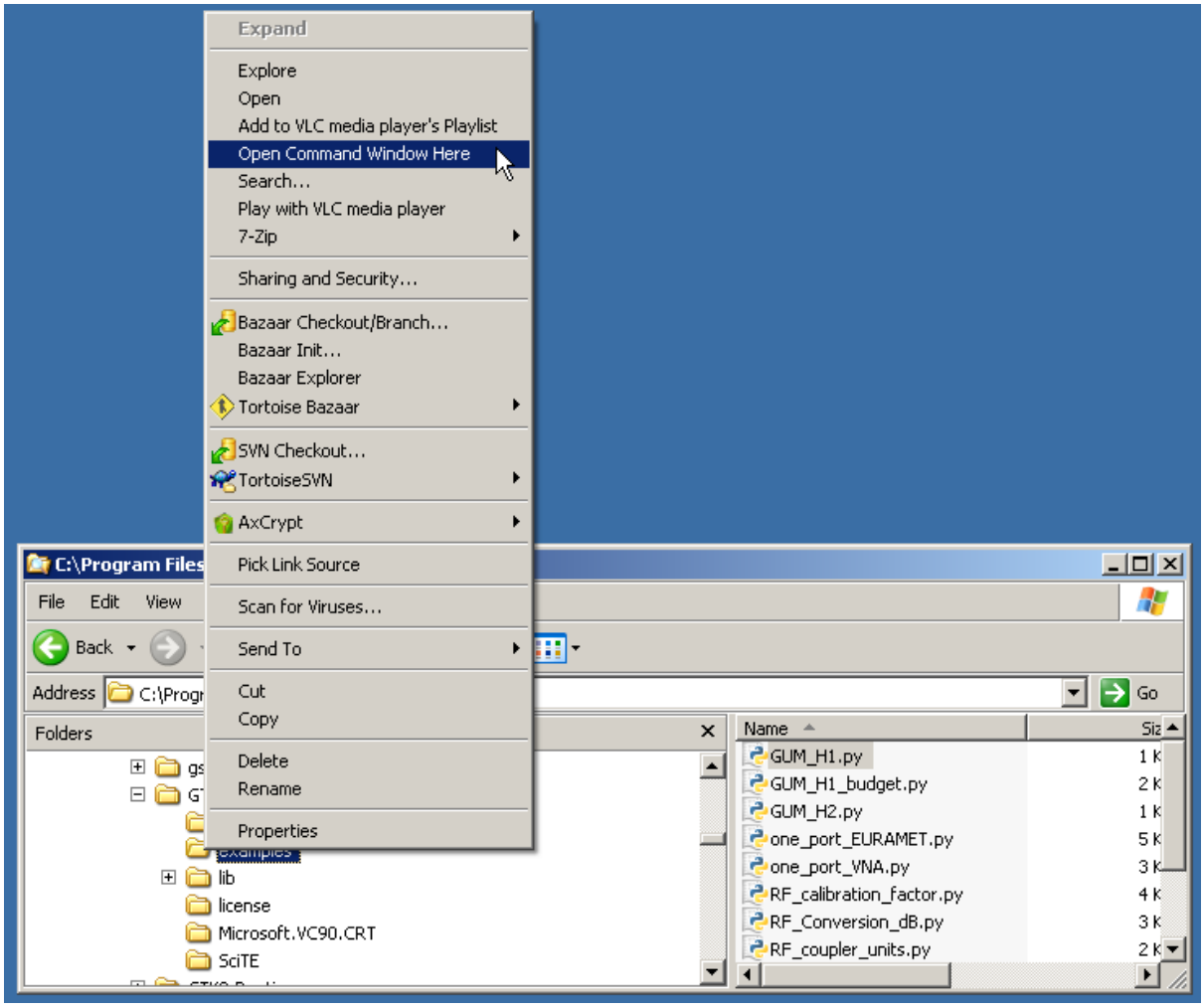
To leave the Help facility, type: `CTRL-C`.

3.2 Explorer context menus

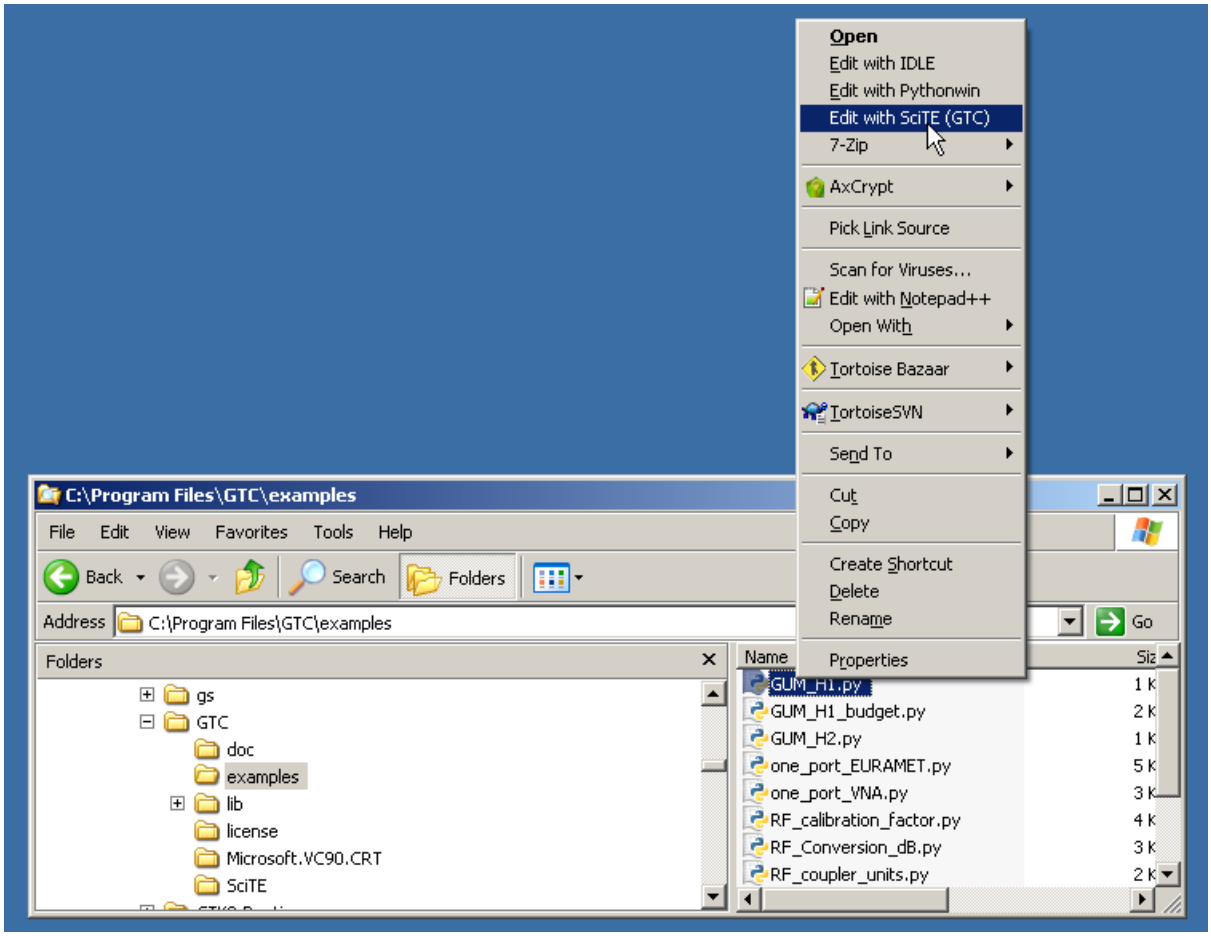
Two options are available in the 'extended' Explorer context menu. A `GTC` item has also been added to the Explorer `SendTo` facility.

Extended context menu items for folders and files

Pressing the `Shift` key while right-clicking on a file or folder in the right-hand pane of the Windows Explorer activates the 'extended' context menu.

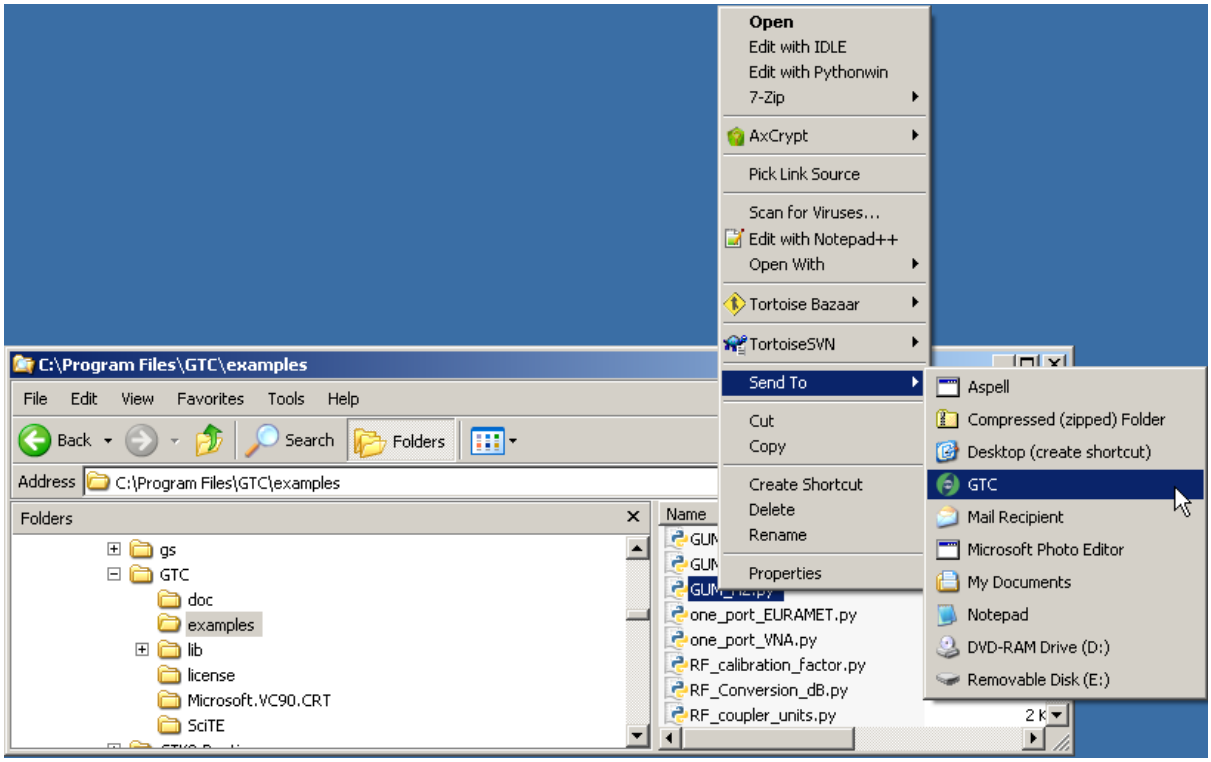


1. When a **folder** is selected, the menu item `Open command window here` opens a Windows Command Prompt in the folder (see *command_line*).
2. When a **file** with a `.py` extension is selected, the `Edit with SciTE (GTC)` option will appear (see *scite_editor*).

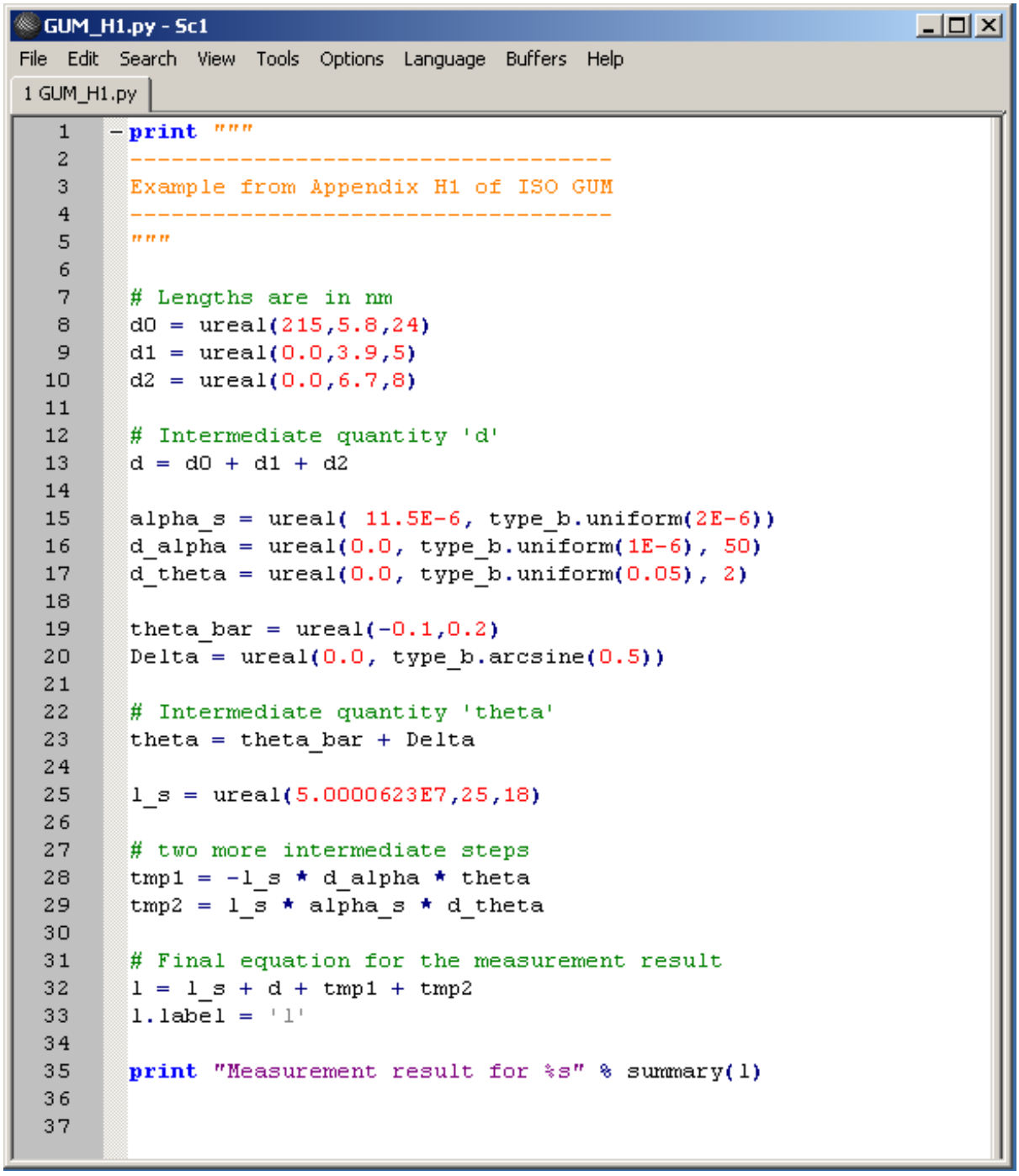


Context menu `SendTo`

The `SendTo` item in the Windows Explorer context menu passes a file to `GTC` for execution. After processing, `GTC` will remain in interactive mode (see *command_line*).


Open
Edit with IDLE
Edit with Pythonwin
7-Zip
AxCrypt
Pick Link Source
Scan for Viruses...
Edit with Notepad++
Open With
Tortoise Bazaar
TortoiseSVN
Send To
Cut
Copy
Create Shortcut
Delete
Rename
Properties
Aspell
Compressed (zipped) Folder
Desktop (create shortcut)
GTC
Mail Recipient
Microsoft Photo Editor
My Documents
Notepad
DVD-RAM Drive (D:)
Removable Disk (E:)
C:\Program Files\GTC\examples
File Edit View Favorites Tools Help
Back Search Folders
Address C:\Program Files\GTC\examples
Folders
gs
GTC
doc
examples
lib
license
Microsoft.VC90.CRT
SciTE
Name
one_port_EURAMET.py
one_port_VNA.py
RF_calibration_factor.py
RF_Conversion_dB.py
RF_coupler_units.py
2 K

3.3 The SciTE editor



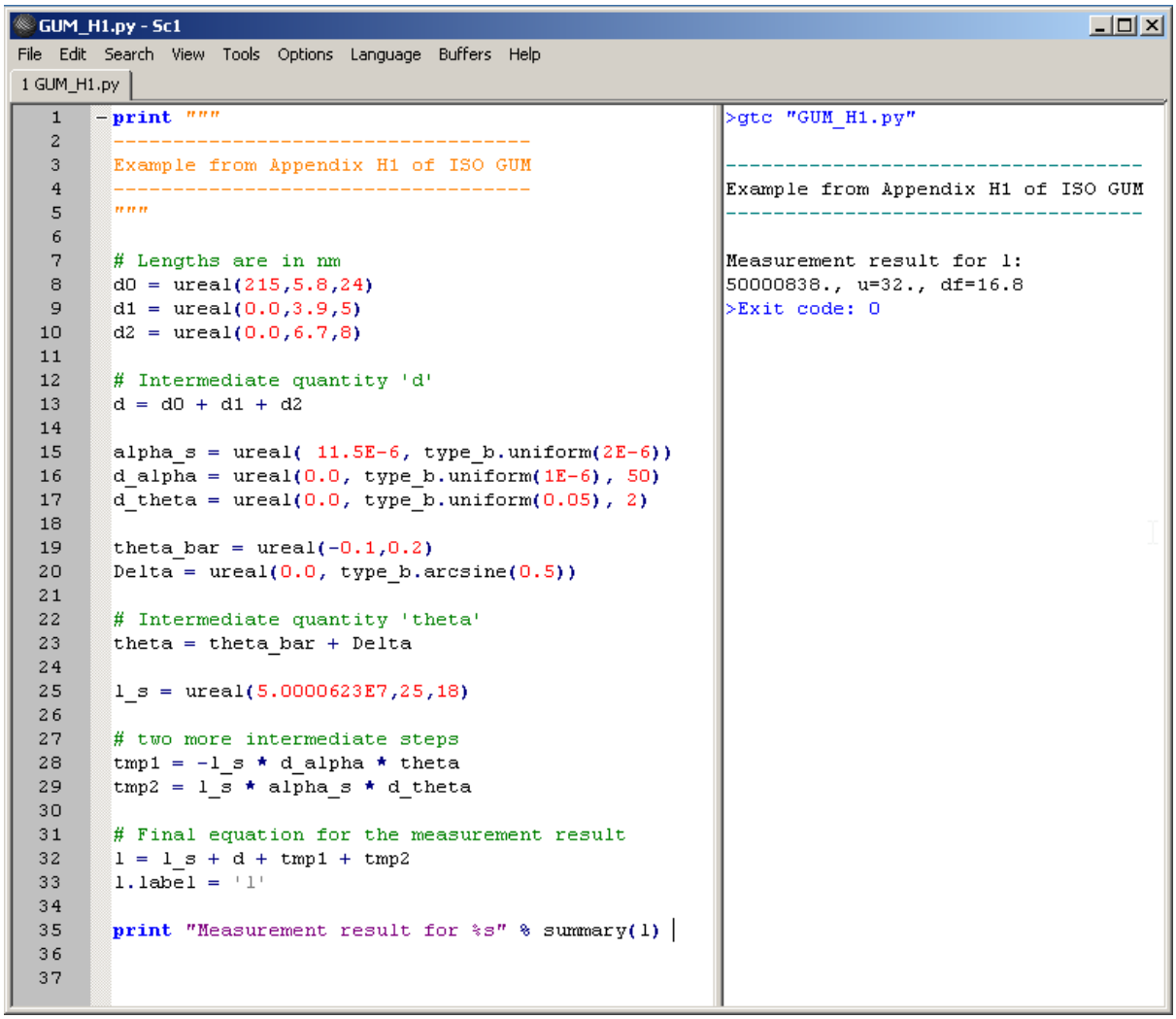
The SciTE editor can be activated in two ways:

- There is a link to SciTE in the `GTC` Start Menu.
- In the Windows Explorer, pressing `SHIFT` while right-clicking on a file with the `.py` extension activates a context menu with an item `Edit with SciTE (GTC)`.

SciTE editor will highlight the language syntax. It has a command-completion feature that provides information about commands as they are typed in. The `GTC` and Python Help files are also integrated.

Executing scripts

SciTE can be used to write and execute scripts:



- The `Go` command (function key `F5`, or menu selection `Tools | Go`) executes the current file.
- The `Interactive Mode` (press `CTRL-3` or select the `Tools | Interactive Mode` menu item) executes the current file, leaving the interpreter in interactive mode. Commands can then be entered at the >>> prompt in the SciTE output window.

  To close an interactive session in SciTE, type:

  ```
  quit()
  ```

  (`CTRL-Z` does not work in SciTE)

Note: Commands can be typed at the >>> prompt in the SciTE output window. However, basic editing features (such as a recalling previous command, command completion, etc) are not available.

In particular, it is impossible to correct any typing mistakes by pressing the `Backspace` key. Doing so erases characters from the display, but it does not remove them from the editor's buffer.

3.3.1 Help inside SciTE

Pressing `F1` opens the `GTC` help file.

Typing `CTRL-4` opens the Python help file.

If a word is selected before pressing either of these help keys, the Help file index is automatically searched for a reference to the word.

CHAPTER

FOUR

A QUICK TOUR

4.1 First steps

In the interactive mode, commands are typed at the prompt. For example,

```
>>> 3 + 3**2
12
```

or

```
>>> print "Hello world"
Hello world
```

Variables can be defined:

```
>>> name = "Mick"
>>> surname = "Dundee"
>>> print name,surname
Mick Dundee

>>> x = 4
>>> y = 2.5
>>> x * y
10.0
```

Iteration is possible:

```
>>> for i in range(10):
...     print i, 2.5 * i
...
0 0.0
1 2.5
2 5.0
3 7.5
4 10.0
5 12.5
6 15.0
7 17.5
8 20.0
9 22.5
```

Note:

- The prompt changes from >>> to ... when a command is incomplete, allowing multi-line statements to be entered, like the `for` statement above.
- Indentation delimits a block of instructions. In the `for` statement above, the second line is indented, making it part of the for-loop body. Any number of lines can be indented to form a block of code.

Conditional statements also use blocks:

```
>>> data = 15
>>> if data > 10:
...     print "big"
... else:
...     print "small"
...
big
```

4.2 Uncertain numbers

GTC uses an *uncertain number* to represent a value that is not precisely known.

Different types of uncertain number are used for real and complex values.

4.2.1 Uncertain real numbers

At least two pieces of information are needed to define an uncertain real number:

- a *value* (the estimate)
- a *standard uncertainty* (of the estimate).

For example, suppose the current flowing in an electrical circuit I and the voltage across a circuit element V have been measured.

The estimates are $V \approx 0.1\,\mathrm{V}$ and $I \approx 15\,\mathrm{mA}$, with standard uncertainties $u(V) = 1\,\mathrm{mV}$ and $u(I) = 0.5\,\mathrm{mA}$, respectively.

Uncertain numbers for V and I are defined using *ureal*

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

The resistance can be calculated using Ohm's law

```
>>> R = V/I
```

Uncertain number attributes

Information about the value (estimate) and the associated uncertainty can be obtained from an uncertain number in different ways.

- Typing the name of an uncertain number at the prompt displays its Python *representation*:

  ```
  >>> R
  ureal(6.666666666666667, 0.23200681130912335, inf)
  ```

  The value is `6.666666666666667` and the standard uncertainty `0.23200681130912335` (`inf` signifies infinite degrees-of-freedom, see note below).

- Attributes, or corresponding functions, can be used to obtain the value, or the uncertainty, etc,

  ```
  >>> R.x
  6.666666666666667
  >>> value(R)
  6.666666666666667

  >>> R.u
  0.23200681130912335
  >>> uncertainty(R)
  0.23200681130912335
  ```

- Alternatively, a summary string can be generated showing the value, the uncertainty and the degrees of freedom. Numbers in the string are formatted to show only significant digits:

  ```
  >>> print R.s
  6.67, u=0.23, df=inf
  >>> summary(R)
  '6.67, u=0.23, df=inf'
  ```

Note: By default, the number of degrees of freedom is infinity (`inf`): this implies that the standard uncertainty is known exactly.

4.2.2 Uncertain complex numbers

To define an uncertain number for a complex quantity, at least two pieces of information are needed:

- a value (the estimate)
- an uncertainty (of the estimate - different formats may be used, see below)

The same functions, or attributes, mentioned above, can be used to obtain information about uncertain complex numbers.

For example, suppose the alternating current flowing in an electrical circuit i, the voltage across a circuit element v and the phase of the voltage with respect to the current ϕ have been measured.

The estimates are: $v \approx 4.999\,\mathrm{V}$, $i \approx 19.661\,\mathrm{mA}$ and $\phi \approx 1.04446\,\mathrm{rad}$, with standard uncertainties $u(v) = 0.0032\,\mathrm{V}$, $u(i) = 0.0095\,\mathrm{mA}$ and $u(\phi) = 0.00075\,\mathrm{rad}$.

Uncertain numbers for v, i and ϕ can be defined using *ucomplex*:

```
>>> 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 that in these definitions the second argument is a pair of numbers [1]. These are the standard uncertainties associated with estimates of the real and imaginary components.

The complex impedance is

```
>>> z = v * exp(phi) / i
```

The value, uncertainty and degrees-of-freedom of `z` are (in summary form)

```
>>> print z.s
(127.73-219.85j), u=[0.19,0.20], r=0.058, df=inf
```

In this format, the complex estimate is followed by the standard uncertainty of the real and imaginary components. The term `r=0.058` is the correlation coefficient between the real and imaginary component estimates.

If the magnitude and phase of z are of interest

```
>>> print magnitude(z).s
254.26, u=0.20, df=inf

>>> print phase(z).s
1.04446, u=0.00075, df=inf
```

Different formats for the uncertainty of uncertain complex numbers

It is possible to specify the uncertainty of a complex estimate in different ways when defining an uncertain complex number:

- When a single number is used, the standard uncertainty of real and imaginary components are equal,
- A 2-element sequence of numbers specifies the standard uncertainties of the real and imaginary components
- A 4-element sequence of numbers specifies the elements of the variance-covariance matrix

  If `v` is such a 4-element sequence then

  - `v[0]` is the standard variance associated with the real component (the standard uncertainty is `math.sqrt(v[0])`)
  - `v[3]` is the standard variance associated with the imaginary component (the standard uncertainty is `math.sqrt(v[3])`)
  - `v[1]` and `v[2]` must be equal, they represent the covariance between the real and imaginary components (the correlation coefficient is `v[1]/math.sqrt(v[0]*v[3])`)

For example,

```
>>> z = ucomplex(1+1j,1)
>>> uncertainty(z)
standard_uncertainty(real=1.0, imag=1.0)
>>> variance(z)
variance_covariance(rr=1.0, ri=0.0, ir=0.0, ii=1.0)

>>> z = ucomplex(1+1j,(.5,.5))
>>> uncertainty(z)
standard_uncertainty(real=0.5, imag=0.5)
>>> variance(z)
```

[1] The parentheses around these numbers are important: they define a type of Python sequence called a *tuple*.

```
variance_covariance(rr=0.25, ri=0.0, ir=0.0, ii=0.25)

>>> z = ucomplex(1+1j,(.5,0.1,0.1,.5))
>>> uncertainty(z)
standard_uncertainty(real=0.70710678118654757, imag=0.70710678118654757)
>>> z = ucomplex(1+1j,(.5,0.1,0.1,.5))
>>> variance(z)
variance_covariance(rr=0.50000000000000011, ri=0.099999999999999992,
    ir=0.099999999999999992, ii=0.50000000000000011)
```

Note: A *namedtuple* is returned by the functions `uncertainty` and `variance`.

The elements of a namedtuple can be accessed by index, or by name. For example,

```
>>> cv = variance(z)
>>> cv
variance_covariance(rr=0.5000000000000001, ri=0.09999999999999999,
    ir=0.09999999999999999, ii=0.5000000000000001)
>>> cv[0]
0.50000000000000011
>>> cv.rr
0.50000000000000011
```

4.3 Programming

GTC uses the Python programming language. This section gives a very brief introduction to some aspects of the language (see references [2] and [3] for more comprehensive Python tutorials).

4.3.1 Sequences

A sequence is a collection of objects. The two main types of sequence are *list* and *tuple*. Tuples cannot be altered (they are read-only) whereas elements may be inserted, changed, sorted, etc, in lists.

For example, a tuple containing the numbers 1 and 2 is

```
>>> tup = (1,2)
```

The elements can be accessed by index (sequences are always base-0)

```
>>> print tup[0]
1
```

[2] *"A Non-Programmer's Tutorial"* <http://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python_2.6>

[3] A list of beginner guides for non-programmers is available here: http://wiki.python.org/moin/BeginnersGuide/NonProgrammers

but cannot not changed

```
>>> tup[1] = 3
Traceback (most recent call last):

  File "<console>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment
```

Lists are created using square brackets

```
>>> l = [-8,6,9]
>>> print l[1]
6
```

The elements of both tuples and lists can be iterated over using `for` loops

```
>>> data = (1.1,3.2,6.7)
>>> for d in data:
...     print d**2
...
1.21
10.24
44.89
```

The function `range` creates a list of integers, which can be useful for iteration

```
>>> for i in range(4):
...     print i
...
0
1
2
3

>>> print range(4)
[0, 1, 2, 3]
```

A useful feature allows sequences to be packed, and unpacked, by matching the number of elements on either side of an = sign. For example,

```
>>> a = 1,2,3
>>> a
(1,2,3)

>>> x,y,z = a
>>> y
2
```

4.3.2 Functions

A function is defined by a name followed by a list of arguments in parentheses and a colon. An indented block of code defines the function body. A `return` statement sends a result back to the calling context.

Here is a function that calculates the surface area of a box, given the length, width and height of the sides

```
def surface(l,w,h):
    end_area = 2 * w * h
    side_area = 2 * l * h
    top_bottom = 2 * w * l
```

```
    area = end_area + side_area + top_bottom

    return area
```

Note: This code snippet would be stored in a file, so the interactive prompt >>> does not appear.

Note: The arguments `l`, `w` and `h` can be ordinary numbers or uncertain numbers.

4.3.3 Strings and printing

When a `print` statement is used to display an uncertain number, the value will be displayed between question marks. The number of significant digits is determined by the uncertainty

```
>>> x = ureal(1.11111,0.1)
>>> print x
?1.11?
```

However, typing the name of the uncertain number (without using `print`) produces a more detailed description, called the Python *representation*:

```
>>> x
ureal(1.111, 0.10000000000000001, inf)
```

There is a way of formatting data as strings (described in detail under *% formatting* in the Python documentation) that uses a template string containing format specifiers to determine how data will appear in a resulting string.

For example, the `%s` format specifier is a placeholder for a string in the output

```
>>> name = 'Jim'
>>> "Hello %s" % name
'Hello Jim'
```

Note: Strings are delimited by apostrophes `'` or quotation marks `"` and multi-line strings begin and end with triple delimiters of either kind.

There are format specifiers for display floating point numbers (e.g., `%f`, `%E`, `%G`)

```
>>> p = math.pi
>>> "pi = %f" % p
'pi = 3.141593'
>>> "pi = %E" % p
'pi = 3.141593E+00'
>>> "pi = %G" % p
'pi = 3.14159'
```

The `%G` format specifier switches automatically to an exponential scientific format when the number becomes too big, or too small (use `%E` to always get the scientific format)

```
>>> bigger = p * 1E6
>>> "pi = %G" % bigger
'pi = 3.14159E+06'
```

Several arguments can inserted in a string using format specifiers. The corresponding data must be contained in a tuple to the right of the `%` operator. For example,

```
>>> quantity = 'voltage'
>>> value = 1.4
>>> unit = 'V'
>>> "%s = %f (%s)" % (quantity,value,unit)
'voltage = 1.400000 (V)'
```

4.3.4 Operators

A few Python operators may be unfamiliar. For example, raising one number to the power of another uses a double asterisk operator

```
>>> 2 ** 4
16
```

Testing for equality uses a double equals operator

```
>>> 2.0 == 2
True
```

Note that a single '=' is used to assign one thing to another, so the following is an error

```
>>> 2.0 = 2
  File "<console>", line 1
SyntaxError: can't assign to literal
```

There are quite a few different operators in Python, so it is best to consult the Python Help file, or a Python reference, for further details [4].

4.3.5 Modules

Libraries of functions and classes can be defined in files called *modules*. Many standard Python modules are included with GTC. For example, *math* and *cmath* define basic mathematical operations for real and complex numbers.

GTC has a module structure of its own. The different GTC modules are all accessible by name (or through a shorter alias). For example, the function *reporting.budget*, which is part of the *reporting* module (alias rp) is useful for obtaining an uncertainty budget, as this short example shows:

```
x1 = ureal(1,0.1,label='x1')
x2 = ureal(1,0.1,label='x2')
x3 = ureal(1,0.1,label='x3')

x4 = x1 + x2
x5 = x2 + x3

x6 = x4 + x5

print "u(x6) = %G" % uncertainty(x6)
for u_cpt in reporting.budget(x6):
    print " %s: %G" % u_cpt
```

The results are

```
u(x6) = 0.244949
 x2: 0.2
 x1: 0.1
 x3: 0.1
```

[4] The Python Help file distributed with GTC can be used to quickly check if a symbol is in fact a Python operator. Open the Help file and type the symbol (or symbols) into the index line. If the symbol is part of Python, the Help file will provide some reference information.

4.3.6 Errors

When programming errors occur, an `exception` is raised. This halts execution and generates an error message called a *traceback*.

Example: RuntimeError

An `exception` is raised when an attempt is made to define an uncertain number with a negative uncertainty

```
>>> x = ureal(1,-1)
Traceback (most recent call last):

  File "<console>", line 1, in <module>

RuntimeError: invalid uncertainty: -1
```

The error message is `invalid uncertainty: -1` and the exception is an instance of a `RuntimeError`. The problem occurred while executing the `"<console>"` file, which is actually the `GTC` command window.

If a file containing the same faulty command had been executed, the error message would look slightly different. Here, executing a file called `simple_error.py`, containing `x = ureal(1,-1)` in line 1, we see:

```
C:\>gtc simple_error.py
Traceback (most recent call last):

  File "simple_error.py", line 1, in <module>
    x = ureal(1,-1)

RuntimeError: invalid uncertainty: -1
```

The first line now refers to the file by name and identifies the code that caused the problem. The `RuntimeError` error message `invalid uncertainty: -1` is given at the bottom, as before.

Other types of error

A `RuntimeError` is just one of a number of built-in Python exceptions. However, the basic structure of traceback messages remain the same.

Here is an example where a `ValueError` is raised while performing a mathematical operation:

```
>>> x = ureal(-1,1)
>>> sqrt(x)
Traceback (most recent call last):

  File "<console>", line 1, in <module>

ValueError: math domain error
```

The exception was raised while executing `sqrt(x)` and the problem is signaled as a `math domain error`.

Part III

User Guide

CHAPTER

FIVE

OVERVIEW

The GUM Tree calculator (`GTC`) is a tool for data processing with automatic uncertainty calculation. It uses *uncertain numbers* to represent quantities that have been measured, or estimated in some way.

5.1 Measurement errors and uncertainty

Measurement can never provide an exact value for a quantity of interest (the *measurand*). The difference, between the true, but unknown, value Y and the measurement result y, is called the *measurement error*

$$E = y - Y \ .$$

When considering the uncertainty of y as an estimate of Y, the likely magnitude of E must be taken into account. Although E is never known, a statistical distribution can be used to describe it. The standard deviation of this distribution is called the standard uncertainty.

5.1.1 Measurement functions

A measurand is often defined in terms of other quantities in an equation

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

where the X_i are called influence quantities.

However, $X_1, X_2, \cdots$ are unknown; only estimates $x_1, x_2, \cdots$ are available. These estimates can be used to calculated an estimate of the measurand

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

The error in y depends on the individual measurement errors $E_1 = x_1 - X_1$, $E_2 = x_2 - X_2$, etc, which are unknown. So, the uncertainty of the result as an estimate of the measurand must be calculated from information about the uncertainties of x_1, x_2, etc.

5.2 Uncertain Numbers

An uncertain number is used to represent a quantity that is unknown. It holds an estimate of that quantity and the uncertainty of that estimate.

Suppose a flag is flying from a pole that is estimated to be 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). The measurement equation is

$$H = B \tan \Phi \ ,$$

where H is the actual height, B is the length from the base of the pole and Φ is the actual line-of-sight angle. The question is: how high is the flag?

The following calculation obtains 11.7 metres, with a standard uncertainty of 0.8 metres

```
>>> B = ureal(15,3E-2)
>>> Phi = ureal(math.radians(38),math.radians(2))
>>> H = B * tan(Phi)
>>> H
ureal(11.719284397600761, 0.843532951107579, inf)
```

It is important to note that GTC calculations are open ended. It is possible to continue the calculation above and evaluate the observer angle at 20 metres from the pole (the base distance still has an uncertainty of 3 cm)

```
>>> B_20 = ureal(20,3E-2)
>>> Phi_20 = atan( H/B_20 )
>>> Phi_20
ureal(0.5300351420781763, 0.031403340387013895, inf)
>>> math.degrees( Phi_20.x )
30.36877663469645
>>> math.degrees( Phi_20.u )
1.7992788667886215
```

5.2.1 Elementary uncertain numbers

We use the term *elementary uncertain number* to describe uncertain numbers associated with problem inputs (e.g., B and Phi above). Elementary uncertain numbers are defined by functions like *ureal* and *ucomplex*.

5.2.2 Uncertain Number Attributes

Uncertain numbers use attributes to provide access to the value (the estimate), the uncertainty (of the estimate) and the degrees of freedom (associated with the uncertainty), as well as some other properties (see *UncertainReal*).

Continuing with the flagpole example, the attributes x, u, df can be used to see the estimate, the uncertainty and the degrees-of-freedom (which is infinity), respectively

```
>>> H.x
11.719284397600761
>>> H.u
0.84353295110757898
>>> H.df
inf
```

Alternatively, there are GTC functions that return the same numbers

```
>>> value(H)
11.719284397600761
>>> uncertainty(H)
0.84353295110757898
>>> dof(H)
inf
```

5.2.3 Uncertain numbers and measurement errors

To make the best use of GTC it is helpful to think in terms of the quantities that appear in measurement equations. Many of these will be residual errors with estimates of zero or unity.

In the context of the example above, the measured data are $b = 15\,\mathrm{m}$ and $\phi = 38\,\mathrm{deg}$, so now let E_b and E_ϕ stand for the respective measurement errors. That is,

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

These are residual errors: our best estimates are $E_b \approx 0$ and $E_\phi \approx 0$, with uncertainties in these estimates of $u(E_b) = 3 \times 10^2\,\mathrm{m}$ and $u(E_\phi) = 2\,\mathrm{deg}$.

The GTC calculation now looks like this

```
>>> b = 15
>>> E_b = ureal(0,3E-2)
>>> B = b - E_b
>>> phi = math.radians(38)
>>> E_phi = ureal(0,math.radians(2))
>>> Phi = phi - E_phi
>>> H = B * tan(Phi)
>>> H
ureal(11.719284397600761, 0.843532951107579, inf)
```

This calculation better reflects our understanding of the problem: $b = 15$ and $\phi = 38$ are precisely known numbers, there is nothing 'uncertain' about their values. The unknown errors E_b and E_ϕ give rise to the uncertainty.

Measurements are sometimes easier to analyse by making the errors explicit in this way.

CHAPTER

SIX

EXAMPLES

Note: There is a shortcut to the folder where the following example files are located in the Start Menu `GTC` group.

6.1 GUM Appendices

6.1.1 Gauge block measurement (GUM H1)

The example described here is taken from Appendix H1 of the GUM [1].

A copy of this script may be found under the user's folder `My GTC\examples\GUM_H1.py`.

```
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')
```

[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

```
# 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 = l_s + d - (tmp1 + tmp2)
l.label = 'l'

print "Measurement result for %s" % summary(l)
```


Explanation

The measurand is the length, at 20 °C, of an end-gauge that is being calibrated. 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 can be described in stages.

First, three inputs are defined: an estimate of the length difference measured by the comparator (`d0`), which is based of the arithmetic mean of a number of difference indications; an estimate of random comparator errors (`d1`) and an estimate of systematic comparator errors (`d2`). These are combined to obtain an 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
```

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

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

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

where

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

In defining the inputs, the variability of some errors is described by uniform or arc-sine distributions. The `GTC` functions *`type_b.uniform`* and *`type_b.arcsine`* are used to convert the width of a particular distribution into a standard uncertainty [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
```

An estimate of 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

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

print "Measurement result for %s" % summary(l)
```

When this script is executed the output is

```
Measurement result for l: 50000838., u=32., df=16.8
```

Uncertainty budget

An uncertainty budget can be obtained.

In interactive mode, after running the script above, the following commands print the components of uncertainty for `l`, due to each influence:

```
print """
Components of uncertainty in l (nm)
-----------------------------------"""
for u_cpt in reporting.budget(l,trim=0):
    print "  %s: %G" % u_cpt
```

The output is

```
Components of uncertainty in l (nm)
-----------------------------------
  ls: 25
```

[3] `ureal` creates a new uncertain real number. It takes a standard uncertainty as its second argument, which is usually the standard deviation of the error distribution.

```
  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
```

Second-order contributions to the uncertainty

The GUM, in H.1.7, notes that the uncertainty associated with the products $\alpha_s \, \delta_\theta$ and $\delta_\alpha \, \theta$ may be underestimated, because in each case one of the factors is estimated as zero.

The GTC calculation can include second-order terms associated with a zero product. This can be done by modifying the definitions of tmp and tmp2 as follows (*function.mul2* includes second-order contributions to the product uncertainty):

```
# two more intermediate steps
tmp1 = l_s * function.mul2(d_alpha, theta)
tmp2 = l_s * function.mul2(alpha_s, d_theta)
```

The result is now

```
Measurement result for l: 50000838., u=34., df=21.6
```

which agrees with the GUM (although no value is given in the GUM for the degrees of freedom).

We note, however, that α_s and θ represent estimates based on measured data. In that case, a different second-order calculation is preferred

```
tmp1 = l_s * function.mul2(d_alpha, theta, estimate=True)
tmp2 = l_s * function.mul2(alpha_s, d_theta estimate=True)
```

which gives

```
Measurement result for l: 50000838., u=32., df=16.5
```

This result is better than the GUM's treatment of this problem, which over-estimates the uncertainty [4].

6.1.2 Resistance and reactance measurement (GUM H2)

The example is taken from Appendix H2 of the GUM [1].

This script, which evaluates the measurement uncertainty, is under the user's folder: My GTC\examples\GUM_H2.py.

```
print """
-------------------------------
Example from Appendix H2 of GUM
-------------------------------
"""

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

[4] B D Hall, *Using simulation to check uncertainty calculations*, Meas. Sci. Technol., **22** (2011) 025105 (10pp) http://mst.irl.cri.nz

[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

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

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

print 'R = %s' % R.s
print 'X = %s' % X.s
print 'Z = %s' % Z.s

print 'Correlation between R and X = %.2G' % get_correlation(R,X)
print 'Correlation between R and Z = %.2G' % get_correlation(R,Z)
print 'Correlation between X and Z = %.2G' % 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. Estimates 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 `dependent=True` is needed to allow the `set_correlation` command to be used).

```
V = ureal(4.999,3.2E-3,dependent=True)       # volt
I = ureal(19.661E-3,9.5E-6,dependent=True)   # amp
phi = ureal(1.04446,7.5E-4,dependent=True)   # 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 = V * cos(phi) / I
X = V * sin(phi) / I
Z = V / I
```

Results are displayed here using the attribute '`.s`' (equivalent to the function *summary*) and the function *get_correlation*:

```
print 'R = %s' % R.s
print 'X = %s' % X.s
print 'Z = %s' % Z.s

print 'Correlation between R and X = %.2G' % get_correlation(R,X)
print 'Correlation between R and Z = %.2G' % get_correlation(R,Z)
print 'Correlation between X and Z = %.2G' % get_correlation(X,Z)
```

The output is

```
R = 127.732, u=0.070, df=nan
X = 219.85, u=0.30, df=nan
Z = 254.26, u=0.24, df=nan
Correlation between R and X = -0.59
Correlation between R and Z = -0.49
Correlation between X and Z = 0.99
```

Calculating the expanded uncertainty

The expanded uncertainty of the results obtained for `R`, `X` and `Z` cannot be evaluated in the GUM, because the Welch-Satterthwaite equation for the effective degrees of freedom is invalid when input estimates are correlated (the reported value of degrees-of-freedom for `R`, `X` and `Z` shown above is `nan`, which signifies an invalid calculation).

However, an alternative calculation is applicable in this case [2].

There are two different ways to carry out the calculation. One uses the `GTC` function *type_a.multi_estimate_real*, the other uses *multiple_ureal*.

`multiple_ureal` allows several elementary uncertain real numbers to be created at the same time and grouped for the purposes of subsequent uncertainty calculations. The documentation for *multiple_ureal* shows how this can be applied to the GUM H2 example.

With *type_a.multi_estimate_real*, a type-A analysis is performed on the raw data (in this case, three sets of five readings) and a set of elementary uncertain real numbers is created. Again, these are identified as a group by `GTC` for the purposes of subsequent calculations. The documentation for *type_a.multi_estimate_real* shows how this is applied to the GUM H2 example.

Note: The impedance calculation can also be treated as a complex-valued problem. There are two other functions in `GTC` that do data processing using uncertain complex numbers. The documentation for *type_a.multi_estimate_complex* and *multiple_ucomplex* both use GUM H2 as an example.

6.1.3 Calibration of a thermometer (GUM H3)

The example described here is taken from Appendix H3 of the GUM [1].

This script, which evaluates the measurement uncertainty, is under the user's folder: `My GTC\examples\GUM_H3.py`.

```
print """
-------------------------------
```

[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

```
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)

# 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
y_1, y_2 = type_a.line_fit(t_rel,b).a_b

# Show the results
print "Intercept: %G u=%G, dof=%G" % (y_1.x,y_1.u,y_1.df)
print "Slope: %G u=%G, dof=%G" % (y_2.x,y_2.u,y_2.df)
print "Correlation r=%G" % get_correlation(y_1,y_2)
print

# Apply correction at 30 C
b_30 = y_1 + y_2*(30.0 - t_0)

print "Correction at 30 C"
print "b(30)=%G, u=%G, dof=%G" % (b_30.x, b_30.u, b_30.df)
```


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.

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.

A pair of numbers y_1 and y_2 are obtained by least-squares regression on b_k and t_k. The uncertainty in y_1 and y_2, as estimates of Y_1 and Y_2, respectively, is due to random fluctuations (E_k) in the measurement system.

The GTC calculation follows. The data are entered in a pair of sequences, the function *type_a.line_fit* then performs the regression. (The intercept and slope are returned as a pair from the a_b attribute.)

```
# 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
y_1, y_2 = type_a.line_fit(t_rel,b).a_b

# Show the results
print "Intercept: %G u=%G, dof=%G" % (y_1.x,y_1.u,y_1.df)
print "Slope: %G u=%G, dof=%G" % (y_2.x,y_2.u,y_2.df)
print "Correlation r=%G" % get_correlation(y_1,y_2)
print
```

The uncertain numbers y_1 and y_2 can be used to calculate the correction for a reading of 30 C

```
# Apply correction at 30 C
b_30 = y_1 + y_2*(30.0 - t_0)

print "Correction at 30 C"
print "b(30)=%G, u=%G, dof=%G" % (b_30.x, b_30.u, b_30.df)
```

The results agree with the numbers reported in the GUM

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

Intercept: -0.171204 u=0.0028776, dof=9
Slope: 0.0021827 u=0.000667939, dof=9
Correlation r=-0.93043

Correction at 30 C
b(30)=-0.149377, u=0.0041386, dof=9
```

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.

First, a sequence of uncertainties for the respective b_k observations can be used with the function *type_a.line_fit_wls* to obtain the slope and intercept. Alternatively, we can define a sequence of uncertain real numbers representing the b_k data and use the function *function.line_fit*

```
u_b = 0.001 # an arbitrary value, just as an example
y_1, y_2 = fn.line_fit(t_rel,[ureal(b_i,u_b) for b_i in b]).a_b
```

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

Systematic error in the standard temperature

The other situation considered in the GUM involves a systematic error associated with the measurement standard reading: all b_k values are subject to some constant error. A type-A analysis is still required to estimate the uncertainty due to system instability. However, the systematic error cannot be evaluated by the statistical analysis (since it is constant).

One way to handle this in GTC, is to combine type-A and type-B regression analyses.

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

```
e_sys = ureal(0,0.005)  # the value of uncertainty is arbitrary
b_sys = [b_i + e_sys for b_i in b]
y_1_sys, y_2_sys = function.line_fit(t_rel,b_sys).a_b
```

Note that e_sys is defined outside the list and then added to each list element. e_sys represents the error common to all the data. The results for y_1_sys and y_2_sys are

```
>>> y_1_sys
ureal(-0.17120379013134998, 0.005, inf)
>>> y_2_sys
ureal(0.0021826977398872772, 1.6263032587282567e-19, inf)
```

The standard uncertainty in y_2_sys is effectively zero, as expected: an error in the temperature standard shifts all values of b_k by the same amount, so the slope does not change.

Next, the type-A regression analysis is done on the b_k data sequence (this operates only on the values of the uncertain numbers)

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

The results are (as before)

```
>>> y_1_a
ureal(-0.17120379013134998, 0.002877597835159957, 9)
>>> y_2_a
ureal(0.0021826977398872772, 0.0006679387732278324, 9)
```

Finally, the results involving the systematic error and the type-A error are combined

```
y_1 = type_a.merge_components(y_1_a,y_1_sys)
y_2 = type_a.merge_components(y_2_a,y_2_sys)
```

which gives

```
>>> y_1
ureal(-0.17120379013134998, 0.005768931382926761, 145.37964721007148)
>>> y_2
ureal(0.0021826977398872772, 0.0006679387732278324, 9.0)
```

Notice that the estimates have not changed, and the standard uncertainty in the slope has not changed either. However, the standard uncertainty of the intercept has increased due to uncertainty about the systematic error, as described in H.3.6 in the GUM.

6.2 EURACHEM / CITAC Guide Examples

6.2.1 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)

```
>>> V = V_flask + V_rep + V_T
>>> V.label = 'V'
```

[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.

The uncertainty calculation

The concentration calculation is then simply [3]

```
>>> c_Cd = 1000 * m * P / V
>>> print "c_Cd=%G, u=%G" % (c_Cd.x,c_Cd.u)
c_Cd=1002.7, u=0.835199 mg/L
```

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

```
>>> for cp in rp.budget(c_Cd):
...     print " %s: %G" % cp
...
 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 cp in rp.budget(c_Cd,[m,P,V]):
...     print " %s: %G" % cp
...
 V: 0.666525
 m: 0.49995
 P: 0.0578967
```

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

6.2.2 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,
- 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.

[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

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 = 0.3888 + u_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 = 8*M_C + 5*M_H + 4*M_O + M_K
```

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 = 18.64 + uV_T_cal + uV_T_temp
```

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='repeatability')
```

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(0.10213615970679071, 0.00010050072212400463, inf)
```

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

[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

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

```
>>> for cpt in rp.budget(c_NaOH,[m_KHP,P_KHP,M_KHP,V_T,R]):
...         print " %s: %G" % cpt
...
 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 cpt in rp.budget(c_NaOH):
...         print " %s: %G" % cpt
...
 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
```

6.2.3 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,
- 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).

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

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 = 0.3888 + u_lin_gross - u_lin_tare
```

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 = 14.89 + uV_T2_cal + uV_T2_temp
V_T2.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 = 18.64 + uV_T1_cal + uV_T1_temp
V_T1.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 = 8*M_C + 5*M_H + 4*M_O + M_K
```

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 = 15 + uV_HCl_cal + uV_HCl_temp
```

[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 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 summary(c_HCl)
```

resulting in

```
c_HCl: 0.10139, u=0.00018, df=inf
```

The contribution from different influences can be examined

```
for cpt in rp.budget(c_HCl,[m_KHP,P_KHP,M_KHP,V_T1,V_T2,V_HCl,R]):
    print " %s: %G" % cpt
```

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

```
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 full uncertainty budget is obtained by

```
for cpt in rp.budget(c_HCl):
    print " %s: %G" % cpt
```

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

```
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
```

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 summary(c_HCl)

for cpt in rp.budget(c_HCl,[m_KHP,P_KHP,M_KHP,V_T1,V_T2,V_HCl,R]):
    print " %s: %G" % cpt
```

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

```
c_HCl: 0.10139, u=0.00016, df=inf

 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
```

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)
print uncertainty(V_T2)
```

The uncertainties are roughly twice the previous values

```
0.0336883837546
0.0332102393849
```

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)
c_HCl.label = 'c_HCl'
print uncertainty(c_HCl)
```

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

```
0.000320501672797
```

6.2.4 Cadmium released from ceramic-ware (A5)

- *The measurement*

[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.

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.

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.

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

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 = (v_leachate * v_fill * v_reading + v_temp + v_cal)/1000  # L
V_L.label = 'V leachate'
print summary(V_L)
```

giving (in L)

```
V leachate: 0.3303, u=0.0018, df=inf
```

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)

B_0, B_1 = fit.a_b
B_0.label = 'B_0'
B_1.label = 'B_1'

print summary(B_0)
print summary(B_1)
```

This gives

```
B_0: 0.0087, u=0.0029, df=13.0
B_1: 0.2410, u=0.0050, df=13.0
```

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

```
print get_correlation(B_0, B_1)
```

yielding

```
-0.870388279778
```

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 observations 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], label='absorbance' )
c_0.label = 'c_0'
print summary(c_0)
```

giving

```
c_0: 0.260, u=0.018, df=13.0
```

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 = a_dia * a_shape
a_V.label = 'a_V'
print summary(a_V)
```

giving [2]

```
a_V: 5.73, u=0.15, df=inf
```

Temperature effect

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

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

[2] Note there is a mistake in the standard uncertainty quoted in the CITAC Guide $u(a_{\mathrm{V}}) = 0.19$, as can be verified by evaluating $\sqrt{(0.042^2 + 0.146^2)}$.

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
r.label = 'r'
print summary(r)
```

resulting in [3]

```
r: 0.0150, u=0.0014, df=45.2
```

The contribution from the different influences can be examined

```
for cpt in rp.budget(r,[c_0,V_L,a_V,f_acid,f_time,f_temp]):
    print " %s: %G" % cpt
```

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:

```
c_0: 0.00102956
f_temp: 0.00086663
a_V: 0.000398736
V leachate: 8.28714E-05
f_time: 1.29994E-05
f_acid: 1.20084E-05
```

The full uncertainty budget can be obtained by writing

```
for cpt in rp.budget(r,trim=0):
    print " %s: %G" % cpt
```

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

```
absorbance: 0.000928623
f_temp: 0.00086663
B_0: 0.000688685
a_shape: 0.00038292
B_1: 0.000311899
dia: 0.000111189
v_reading: 6.128E-05
v_cal: 4.63764E-05
v_fill: 3.0794E-05
```

[3] The mistake in $u(a_{\text{V}})$, mentioned above , leads to give a slightly different value $u(r) = 0.0015\ \text{mg}\,\text{dm}^{-2}$ in the CITAC Guide.

```
f_time: 1.29994E-05
f_acid: 1.20084E-05
v_temp: 3.65814E-06
```

6.3 Linear calibration

6.3.1 Linear Calibration Equations

This section shows how GTC can be applied to a simple calibration problem [1].

The calibration

A pressure sensor is to be calibrated. The type of sensor is known to have an approximately linear response.

Eleven reference pressures are accurately generated at a calibration laboratory and the corresponding sensor indications are recorded.

The *y_data* are standard pressures and *x_data* are sensor readings (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)
```

We are told that the sensor indication does not change when observations are repeated at the same standard pressure, which suggests that the digital resolution of the sensor is much less than any random system noise. So we ignore noise as a source of error.

The measurement model

A linear model of the sensor's intrinsic behaviour is

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

where Y is the applied pressure and X is a an internal property of the sensor related to the applied pressure. The sensor indication, x is an estimate of X.

In operation, 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 unknown errors.

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

$E_{\mathrm{res}\cdot i}$ is due to the finite number of digits displayed, i.e., the number displayed is $x_i = X_i + E_{\mathrm{res}\cdot i}$. We will assign an uncertainty to x_i as an estimate of X_i below.

$E_{\mathrm{lin}\cdot i}$ is the difference between the applied pressure Y_i and the value of the linear model $\alpha + \beta\, X_i$. $E_{\mathrm{lin}\cdot i}$ is ignored initially, while estimating α and β [2].

The reference pressure $Y_{\mathrm{cal}\cdot i}$ is not known exactly, but the nominal pressure is

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

where $E_{\mathrm{cal}\cdot i}$ is the measurement error. The uncertainty of $y_{\mathrm{cal}\cdot i}$ as an estimate of $Y_{\mathrm{cal}\cdot i}$ has been determined by the calibration laboratory and was reported as a relative standard uncertainty

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

A calibration procedure estimates α and β.

A 2-point calibration curve

The slope of the calibration curve can be found as

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

Points near the ends of the range of data available are most influential when estimating the slope and intercept of a linear calibration function (but the remainder of the calibration data are used later to assess the importance of non-linear sensor response across the range), Hence 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 becomes

```
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

We can now look at the differences between the calibration line and the calibration data points. We can display a table of differences

[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.

```
for x_i,y_i in zip(x_data,y_data):
    dif = y_i - (x_i * b + a)
    print "x=%G, dif=%G" % (x_i,dif)
```

which generates

```
x=0, dif=0
x=0.2039, dif=0.00195982
x=0.408, dif=0.00195982
x=0.612, dif=0.00293974
x=0.816, dif=0.00391965
x=1.0201, dif=0.00391965
x=1.2242, dif=0.00391965
x=1.4283, dif=0.00391965
x=1.6325, dif=0.00293974
x=1.8367, dif=0.00195982
x=2.041, dif=0
```

From this data, a maximum deviation (worst case error) of 0.005 is obtained [1] that accounts for the deviations from linearity of the sensor [3].

The calibration equation

The procedure above obtains sufficient information to define a calibration function that takes a sensor reading and returns an uncertain number corresponding to the pressure estimate.

For instance (assuming that we now have only the results given above, from a calibration report, not the calibration data.),

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

a = ureal(0.0,0.00028,label='a')
b = ureal(9.79912, 0.00114,label='b')
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
```

We can calculate pressure estimates with expanded uncertainties ($k = 2$), as in Table 7 of the reference [1], by applying this function to the calibration data

```
for i,x_i in enumerate(x_data):
    y_i = cal_fn(x_i)
    print "%i: p=%G,  U(p)=%G" % (i,y_i.x,2*y_i.u)
```

The output is

```
0: p=0.0000,  U(p)=0.0058
1: p=1.9980,  U(p)=0.0058
```

[3] A linear model is chosen for simplicity of use by the client. There is an obvious bias in the residuals that is ignored at this stage.

```
2: p=3.9980,  U(p)=0.0059
3: p=5.9971,  U(p)=0.0060
4: p=7.9961,  U(p)=0.0061
5: p=9.9961,  U(p)=0.0062
6: p=11.996,  U(p)=0.0064
7: p=13.996,  U(p)=0.0066
8: p=15.997,  U(p)=0.0069
9: p=17.998,  U(p)=0.0071
10: p=20.000,  U(p)=0.0074
```

Linearising the sensor response

To improve measurements with this type of sensor a function can be used to pre-process readings

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

The c_i coefficients are **not** determined as part of the calibration procedure so no uncertainty need be associated with these numbers.

We implement this function as follows

```
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
```

In operation, our model of the sensor now becomes

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

The effect of f_{lin} is to reduce the difference between the pressure estimates and actual pressures.

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 includes the reading error uncertainty).

```
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, 0.00028307798251305335, inf)
>>> b
ureal(1.000011112006328, 0.00011672745986082041, inf)
>>> get_correlation(a,b)
-0.12125729816056871
```

The differences between nominal standard values and the sensor estimates can be generated 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=%G" % (x_i,dif)
```

We see that the differences are much smaller than before

```
x=0, dif=0
x=0.2039, dif=0.000632846
x=0.408, dif=-0.00047867
x=0.612, dif=-0.000363706
x=0.816, dif=2.65297E-05
x=1.0201, dif=-0.00025863
x=1.2242, dif=-0.000209565
x=1.4283, dif=0.000203072
x=1.6325, dif=2.9212E-05
x=1.8367, dif=0.000278049
x=2.041, dif=0
```

The worst-case error is now about ± 0.0007.

The calibration equation

A new calibration function can be defined that includes `f_lin`. This takes the same raw sensor reading as before, but returns a better estimate of the applied pressure

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

a = ureal(0.0,0.00028,label='a')
b = ureal(1.000011, 0.000117,label='b')
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 points

```
for i,x_i in enumerate(x_data):
    y_i = lin_cal_fn(x_i)
    print "%i: p=%#0.5G,  U(p)=%#.2G" % (i,y_i.x,2*y_i.u)
```

The results are:

```
0: p=0.0000,  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=8.0000,  U(p)=0.0021
5: p=10.000,  U(p)=0.0025
6: p=12.000,  U(p)=0.0030
7: p=14.000,  U(p)=0.0034
8: p=16.000,  U(p)=0.0038
```

```
9: p=18.000,  U(p)=0.0043
10: p=20.000,  U(p)=0.0047
```

6.3.2 Linear Regression Results

Conventional least-squares regression of a line to a set of data provides estimates of 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 stable stimuli, and `y` data that are observations of the system response 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 `fit` (returned by *`type_a.line_fit`*) 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 constants, 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 provides generates the `fit` object above, which holds a pair of uncertain numbers representing α and β:

```
>>> a, b = fit.a_b
>>> a
ureal(3.829633197588695, 1.7684473272506525, 31)
>>> b
ureal(0.9036432105793234, 0.050118973559182003, 31)
>>> get_correlation(a,b)
-0.9481240708919155
```

The mean response

The uncertain numbers `a` and `b` can be used to calculate the mean response for a particular stimulus, say $x = 21.5$

```
>>> y = a + b*21.5
>>> y
ureal(23.25796222504415, 0.8216070588885063, 31.0)
```

y is an estimate of the response that would be observed, in the absence of noise, to a stimulus of 21.5.

A single future response

A single response in the future to a given stimulus may also be of interest (as opposed to the mean response). Again, say $x = 21.5$

```
>>> y0 = fit.y_from_x(21.5)
>>> y0
ureal(23.25796222504415, 3.3324092579571105, 31.0)
```

The value predicted is the same as before, but the uncertainty is much larger now.

Estimating the stimulus from observations of the response

Another possibility is that several observations of the response to a steady stimulus may be used to estimate that stimulus value [1].

Suppose that three observations were made `[31.4,29.3,27.1]`

```
>>> x0 = fit.x_from_y( [31.4, 29.3, 27.1] )
>>> x0
ureal(28.149421332751846, 2.1751408733425195, 31.0)
```

The value of `x0` is an estimate of the stimulus based on the mean of the observations and taking into account the variability of the *y* data used in the regression.

6.3.3 Straight-line calibration functions

This section shows how GTC uses straight-line least-squares regression algorithms to obtain and use 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.

Each example consists of a small sample of *x*-*y* observation pairs, together with information about the variability of the data. No measurement context is given [1].

In some of the examples, we show how the GTC results can be used to estimate either a stimulus value x, when given further observations of the response y, or to evaluate a future response y to a 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 a weighted least-squares regression to the data, which assumes that the values in u_y are known uncertainties for the y data (i.e., they have infinite degrees of freedom)

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

That generates the following

```
Weighted Least-Squares Results:

  Number of points: 6
  Intercept: 1.87, u=0.47, df=inf
  Slope: 1.76, u=0.12, df=inf
  Correlation: -0.9
  Sum of the squared residuals: 1.66476
```

More significant figures can be obtained with these commands

```
a, b = fit.a_b
print "a=%.15G, u=%.15G" % (value(a),uncertainty(a))
print "b=%.15G, u=%.15G" % (value(b),uncertainty(b))
print "cov(1,b)=%.15G" % (a.u*b.u*get_correlation(a,b))
```

giving

```
a=1.86666666666667, u=0.465474668125631
b=1.75714285714286, u=0.119522860933439
cov(1,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) = -0.50
chi-squared = 1.665, with 4 degrees of freedom
```

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

[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.

Application: an additional *y* observation

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

For example, if we observe $y_1 = 10.5$, and $u(y_1) = 0.5$, we can estimate the corresponding stimulus x_1 as follows

```
y1 = ureal(10.5,0.5)
x1 = (y1-a)/b
print "x1=%.15G, u=%.15G" % (value(x1),uncertainty(x1))
```

giving

```
x1=4.91327913279133, u=0.32203556012891
```

The result includes uncertainty in the estimates of slope and intercept, which have been propagated from the uncertain numbers a and b.

Forward evaluation: an additional *x* value

The regression results can also be used to estimate the response y corresponding to a stimulus x.

For example, if $x_2 = 3.5$, and $u(x_2) = 0.2$, we estimate y_2 as follows

```
x2 = ureal(3.5,0.2)
y2 = a + b*x2
print "y2=%.15G, u=%.15G" % (value(y2),uncertainty(y2))
```

giving

```
y2=8.01666666666667, u=0.406409531732455
```

Again, the uncertain number for y_2 includes uncertainty in 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, y and the uncertainties 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. This choice implies that the uncertainties in y values are exactly known (i.e., they have infinite degrees of freedom)

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

This generates

```
Weighted Least-Squares Results:

  Number of points: 6
  Intercept: 0.89, u=0.53, df=inf
  Slope: 2.06, u=0.18, df=inf
  Correlation: -0.87
  Sum of the squared residuals: 4.1308
```

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(1,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, the results for a and b can be used to calculate x when a further observation of y is available. For example, if $y_1 = 10.5$ and $u(y_1) = 1.0$, x_1 is obtained in the same way as Example 1

```
y1 = ureal(10.5,1)
x1 = (y1-a)/b
print "x=%.15G, u=%.15G" % (value(x1),uncertainty(x1))
```

giving

```
x=4.67425641025641, u=0.533180902231294
```

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

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

The data sequences for x, y and the 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 need to use total least-squares regression in this case, because there is uncertainty in both the dependent and independent variables. Weighted least-squares is used initially, to obtain an estimate of the slope and intercept,

```
fit_i = ta.line_fit_wls(x,y,u_y)
print fit_i
```

which produces

```
Weighted Least-Squares Results:

  Number of points: 6
  Intercept: 0.66, u=0.22, df=inf
  Slope: 2.148, u=0.076, df=inf
  Correlation: -0.89
  Sum of the squared residuals: 9.70522
```

The weighted total least-squares regression algorithm uses these estimates of slope and intercept

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

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

giving

```
Weighted Total Least-Squares Results:

  Number of points: 6
  Intercept: 0.58, u=0.48, df=4.0
  Slope: 2.16, u=0.14, df=4.0
  Correlation: -0.9
  Sum of the squared residuals: 2.74268
```

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(1,b)=-0.0586143419560877
```

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
```

(The slight differences are 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 values has been collected. 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 *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
```

which displays

```
Relative Weighted Least-Squares Results:

  Number of points: 6
  Intercept: 1.17, u=0.16, df=4.0
  Slope: 1.964, u=0.041, df=4.0
  Correlation: -0.9
  Sum of the squared residuals: 0.116498
```

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]

[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.

```
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 $u(a)$ and $u(b)$. This is the usual statistical treatment. However, a trend in recent uncertainty guidelines is to dispense with the frequentist statistics notion of degrees of freedom. So, in a final step, reference [1] multiplies $u(a)$ and $u(b)$ by an additional factor of 2. We do not believe that this last step is correct. `GTC` uses the finite degrees of freedom associated with $u(a)$ and $u(b)$ when calculating the coverage factor required for an expanded uncertainty.

Example 5: unknown uncertainty in *y*

The data in previous example could also have been processed with an 'ordinary' least-squares algorithm, because the scale factor for each observation of y was unity. In effect, a series of six pairs of `x-y` observations were collected and the variance associated with each observation was assumed 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
```

which displays

```
Ordinary Least-Squares Results:

  Number of points: 6
  Intercept: 1.17, u=0.16, df=4.0
  Slope: 1.964, u=0.041, df=4.0
  Correlation: -0.9
  Sum of the squared residuals: 0.116498
```

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 that gave rise to a response $y_1 = 10.5$, we can use the object `fit` returned by the regression function (note that *`x_from_y`* takes a sequence of y values)

```
y1 = 10.5
x1 = fit.x_from_y( [y1] )
print summary(x1)
```

which displays

```
4.751, u=0.097, df=4
```

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 summary(y2)
```

giving

```
8.04, u=0.18, df=4
```

In this case, the uncertainty includes 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 can be obtained directly from the fitted parameters

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

which gives

```
8.044, u=0.070, df=4
```

6.4 RF and microwave problems

6.4.1 Mismatch

- *Known complex reflection coefficients*
- *Unknown phases*

Known complex reflection coefficients

An effect called *mismatch* arises frequently in RF and microwave measurements. In the context of a simple power measurement the mismatch correction factor is

$$M = |1 - \Gamma_\mathrm{s}\Gamma_\mathrm{g}|^2 \ ,$$

where Γ_s and Γ_g are complex reflection coefficients associated with a power sensor and a signal generator.

Suppose $\Gamma_\mathrm{s} \approx 0.01 - \mathrm{j}0.02$ and $\Gamma_\mathrm{g} \approx 0.04 - \mathrm{j}0.07$.

The standard uncertainties (for both the real and imaginary components) are $u(\Gamma_\mathrm{s}) = 0.025$ and $u(\Gamma_\mathrm{g}) = 0.05$.

To calculate the mismatch proceed as follows

```
>>> G1 = ucomplex(0.01-0.02j,0.025,label='G1')
>>> G2 = ucomplex(0.04-0.07j,0.05,label='G2')
>>> M = mag_squared(1 - G1*G2)
>>> M.s
'1.0020, u=0.0046, df=inf'
```

The function *reporting.budget* displays the uncertainty budget

```
>>> for l,u in reporting.budget(M): print l,u
...
G1_im 0.0035065
G2_im 0.0020035
G1_re 0.00199675
G2_re 0.000998
```

Because M is real-valued, the uncertainty budget is broken down in terms of the real and imaginary components of the complex influence quantities.

Information about individual uncertainty components can be obtained with the function `component`, for example

```
>>> component(M,G1.imag)
0.0035065000000000001
>>> component(M,G2.imag)
0.0020035000000000001
```

Unknown phases

Often a mismatch term must be calculated without information about the phases of the reflection coefficients. In that case the best estimate of the complex reflection coefficient is zero and the associated uncertainty depends on what is known about the magnitudes of the reflection coefficients.

There are several possibilities:

1. the magnitude is known, $|\Gamma| = a$
2. there is an upper limit on the magnitude $|\Gamma| \leq a$

Functions can be used to transform the information available in each of these cases into a standard uncertainty (see: *uniform_ring* and *uniform_disk*).

In addition, the uncertainty associated with the product $\Gamma_s\Gamma_g$ in the mismatch calculation will not propagate when the estimates are zero (a consequence of the linear approximation always made for uncertainty propagation).

For instance

```
>>> Gs = ucomplex(0,type_b.uniform_ring(0.07),label='Gs')
>>> Gg = ucomplex(0,type_b.uniform_ring(0.05),label='Gg')
>>> M = mag_squared(1 - Gs*Gg)
>>> M
ureal(1.0, 0.0, nan)
```

There are two ways of addressing this using `GTC`. One is to define an elementary uncertain number representing the product $X = \Gamma_s\Gamma_g$ [1]. This can be done with *unknown_phase_product*, as follows

```
>>> us = type_b.uniform_ring(0.07)
>>> ug = type_b.uniform_ring(0.05)
>>> X = ucomplex( 0, type_b.unknown_phase_product(us,ug),label = 'X' )
>>> X
ucomplex( 0j, [6.1250000000000015e-06,0.0,0.0,6.1250000000000015e-06], inf,␣
↪label=X )
```

[1] B D Hall, *On the expression of measurement uncertainty for complex quantities with unknown phase*, Metrologia, **48** (2011) 324-332, on-line: http://mst.irl.cri.nz

```
>>> M = mag_squared(1 - X)
>>> M
ureal(1.0, 0.004949747468305833, inf)
>>> for cpt in rp.budget(M,trim=0):
...         print "  %s: %G" % cpt
...
  X_re: 0.00494975
  X_im: 0
```

Alternatively, *function.mul2* may be used. This implements a second-order extension of the basic uncertainty propagation method

```
>>> X = fn.mul2(Gs,Gg)
>>> M = mag_squared(1 - X)
>>> M
ureal(1.0, 0.0049497474683058325, inf)
>>> for cpt in rp.budget(M,trim=0):
...         print "  %s: %G" % cpt
...
  Gs_re: 0.00247487
  Gs_im: 0.00247487
  Gg_re: 0.00247487
  Gg_im: 0.00247487
```

The second method has the advantage that uncertainty components associated with `Gs` and `Gg` are propagated into `M`. However, the method is non-standard and goes beyond the usual linear approximation associated with uncertain propagation.

Note there are restrictions on the arguments that can be used with *mul2*.

6.4.2 Equivalent reflection coefficient

Direct calculation

There is a well-known expression for the reflection coefficient seen looking back into a linear two-port network when the far port is terminated by something with a reflection coefficient Γ,

$$\Gamma' = S_{11} + \frac{S_{21}S_{12}\Gamma}{1 - S_{22}\Gamma} ,$$

where the S_{ij} are complex-valued *S*-parameters describing the two-port network.

If the values and uncertainty of the *S*-parameters and Γ are known, we can calculate Γ'.

For example

```
>>> S11 = ucomplex( 0.05 - 0.03j, 0.02, label='S11')
>>> S21 = ucomplex( 0.91 - 0.06j, 0.03, label='S21')
>>> S12 = ucomplex( 0.95 - 0.02j, 0.03, label='S12')
>>> S22 = ucomplex( 0.07 + 0.02j, 0.02, label='S22')

>>> Gamma = ucomplex( 0.3+0.2j, (0.02,0.04), label = 'Gamma')
```

the effective reflection coefficient Γ' is

```
>>> G_eff = S11 + (S12 * S21 * Gamma)/ (1 - S22 * Gamma)
>>> value(G_eff)
(0.325548278712+0.128302101296j)
>>> uncertainty(G_eff)
standard_uncertainty(real=0.03064091238057904, imag=0.0435781566142131)
```

Using a function

A function that calculates the effective reflection coefficient could be used more conveniently when there is a number of different sets of measurements. One possible implementation is

```
def gamma_equiv(Gamma,S11,S21,S12,S22):
    """Return the effective reflection coefficient
    """
    num = S12 * S21 * Gamma
    den = 1.0 - S22 * Gamma
    return S11 + num/den
```

Here we use the *S*-parameters and Γ defined earlier and generate an uncertainty budget (the numbers displayed are summary values, see *reporting.u_bar*)

```
>>> Gamma_prime = gamma_equiv(Gamma,S11,S21,S12,S22)
>>> print summary(Gamma_prime)
>>> for influence,u_cpt in reporting.budget(Gamma_prime):
...     print influence,':',u_cpt
```

The output is

```
(0.326+0.128j), u=[0.031,0.044], r=0.033, df=inf
Gamma : 0.0283476068202
S11 : 0.02
S21 : 0.0104536840276
S12 : 0.0100330480747
S22 : 0.00233071809794
```

Using a class

If the reflection coefficient is to be calculated repeatedly, while the set of *S*-parameters remains unchanged, a class is also a convenient option

```
class TwoPort(object):
    def __init__(self,S11,S21,S12,S22):
        self.S11 = S11
        self.S12 = S12
        self.S21 = S21
        self.S22 = S22
    def gamma_equiv(self,gamma):
        num = self.S12 * self.S21 * gamma
        den = 1 - self.S22 * gamma
        return self.S11 + num/den
```

`TwoPort` objects calculate the effective reflection coefficient for different values of Γ

```
>>> two_port = TwoPort(S11,S12,S21,S22)
>>> G_eff = two_port.gamma_equiv(Gamma)
>>> value(G_eff)
(0.325548278712+0.128302101296j)
>>> uncertainty(G_eff)
```

```
standard_uncertainty(real=0.03064091238057904, imag=0.0435781566142131)

>>> Gamma2 = ucomplex( 0.9+0.12j, (0.03,0.03), label = 'Gamma2')
>>> G_eff2 = two_port.gamma_equiv(Gamma2)
>>> print summary(G_eff2)
(0.885+0.032j), u=[0.055,0.055], r=0.0, df=inf
```

6.4.3 One-port vector network analyser calibration

This example involves a typical microwave measurement procedure that corrects for systematic errors in a vector network analyser (VNA). This is commonly referred to as VNA 'calibration'.

Theory

Calibration of one port of a VNA can be carried out by measuring three standards, each with a nominal value (perhaps from a calibration report). The pairs of measured and nominal values are used to calculate terms that are used to adjust raw VNA readings for systematic errors during VNA operation.

The correction terms can be found by solving for A, B and C in the following equations

$$\begin{bmatrix} \Gamma_1 & 1 & -\Gamma_1^{\mathrm{m}}\,\Gamma_1 \\ \Gamma_2 & 1 & -\Gamma_2^{\mathrm{m}}\,\Gamma_2 \\ \Gamma_3 & 1 & -\Gamma_3^{\mathrm{m}}\,\Gamma_3 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} \Gamma_1^{\mathrm{m}} \\ \Gamma_2^{\mathrm{m}} \\ \Gamma_3^{\mathrm{m}} \end{bmatrix}$$

where Γ_1, etc, are the nominal reflection coefficients of the standards and Γ_1^{m}, etc, are the corresponding measured values (all complex numbers).

A, B and C can be found by standard methods of linear algebra. In one final step, the conventional VNA error terms are obtained

$$\begin{aligned} E_{\mathrm{D}} &= B\,, \\ E_{\mathrm{S}} &= -C\,, \\ E_{\mathrm{R}} &= A - BC\,. \end{aligned}$$

Calibration software

The calibration script makes use of the function `osl`, defined in `one_port_cal.py` [1].

```
"""
A one-port VNA calibration using uncertain numbers.

This script shows how uncertain number data for both measured
values and nominal values (e.g., from a calibration certificate)
can be used to evaluate the three error terms of a one-port
vector network analyser.
```

[1] A special module *one_port_cal.py*, stored in the `gtc/lib/rf` folder of the installation, is used in this example. *one_port_cal.py* is an example of a *user-defined* extension module for `GTC`.

```
The script uses the function OSL(), defined in another module
(rf.one_port_cal), to evaluate the errors.

The module 'rf.one_port_cal' is the file
'.../gtc/lib/rf/one_port_cal.py', where '.../gtc' means
the folder where gtc has been installed (e.g. /Program Files/gtc).

"""
# BEGIN_preamble
from rf.one_port_cal import osl
# END_preamble

print "Example one-port Open-Short-Load calibration"
print "============================================"
print

# BEGIN_inputs
#-----------------------------------------------------
# Measured data is a sequence of uncertain complex numbers
#
measured = (
    ucomplex(-0.188 - 0.902j,0.05),         # std 1
    ucomplex(0.239 + 0.936j,0.05),         # std 2
    ucomplex(0.006 + 0.007j,0.05),         # std 3
)

#-----------------------------------------------------
# Nominal data is a sequence of uncertain complex numbers
#
nominal = (
    ucomplex(-1 + 0j,0.01),                 # std 1
    ucomplex(1 + 0j,0.01),                 # std 2
    ucomplex(0 + 0j,0.01),                 # std 3
)
# END_inputs

#-----------------------------------------------------
# The errors are calculated by the OSL() function in
# the module 'rf.one_port_cal'
#
# BEGIN_calibration
errors = osl(measured,nominal)
# END_calibration

# BEGIN_error_terms
#-----------------------------------------------------
# Report the results
#
print "One-port errors"
print "---------------"
print 'E_D', summary( errors.E_D )
print 'E_S', summary( errors.E_S )
print 'E_R', summary( errors.E_R )
# END_error_terms

print

# BEGIN_correlations
#-----------------------------------------------------
# The errors are not independent. The most significant
# correlation occurs between the directivity and the
# source match terms.
```

```
#
q = get_correlation(errors.E_D,errors.E_S)

print "Correlation between components of E_S and E_D"
print "---------------------------------------------"
print "  E_S_re and E_D_re :" ,  q.rr
print "  E_S_re and E_D_im :" ,  q.ri
print "  E_S_im and E_D_re :" ,  q.ir
print "  E_S_im and E_D_im :" ,  q.ii
# END_correlations
```

Explanation

The function `osl` is imported from the user-defined library

```
from rf.one_port_cal import osl
```

Next, measured and nominal standards are defined, using *ucomplex*. When a single real number specifies the uncertainty, it means that the standard uncertainties associated with the real and imaginary components are equal

```
#-------------------------------------------------------
# Measured data is a sequence of uncertain complex numbers
#
measured = (
    ucomplex(-0.188 - 0.902j,0.05),         # std 1
    ucomplex(0.239 + 0.936j,0.05),         # std 2
    ucomplex(0.006 + 0.007j,0.05),         # std 3
)

#-------------------------------------------------------
# Nominal data is a sequence of uncertain complex numbers
#
nominal = (
    ucomplex(-1 + 0j,0.01),                 # std 1
    ucomplex(1 + 0j,0.01),                 # std 2
    ucomplex(0 + 0j,0.01),                 # std 3
)
```

The error-correction terms are calculated by `osl`

```
errors = osl(measured,nominal)
```

The results are returned in a `namedtuple` called `errors`.

```
#-------------------------------------------------------
# Report the results
#
print "One-port errors"
print "---------------"
print 'E_D', summary( errors.E_D )
print 'E_S', summary( errors.E_S )
print 'E_R', summary( errors.E_R )
```

Giving the following output:

```
One-port errors
---------------
E_D (0.0060+0.0070j), u=[0.051,0.051], r=0.00, df=inf
E_S (0.015-0.018j), u=[0.066,0.066], r=0.00, df=inf
E_R (0.213+0.919j), u=[0.036,0.036], r=0.00, df=inf
```

There may be correlation between the components of different error terms (the function *get_correlation* returns a `namedtuple` of four elements with attributes: `.rr`, `.ri`, etc.). The most significant correlation coefficient, in this case, is between the directivity and the source-match.

We write

```
#-------------------------------------------------------
# The errors are not independent. The most significant
# correlation occurs between the directivity and the
# source match terms.
#
q = get_correlation(errors.E_D,errors.E_S)

print "Correlation between components of E_S and E_D"
print "---------------------------------------------"
print "  E_S_re and E_D_re :" ,  q.rr
print "  E_S_re and E_D_im :" ,  q.ri
print "  E_S_im and E_D_re :" ,  q.ir
print "  E_S_im and E_D_im :" ,  q.ii
```

which produces the output:

```
Correlation between components of E_S and E_D
---------------------------------------------
  E_S_re and E_D_re : -0.184749680584
  E_S_re and E_D_im : 0.795245697689
  E_S_im and E_D_re : -0.795245697689
  E_S_im and E_D_im : -0.184749680584
```

one_port_cal.py

The extension module `one_port_cal.py` defines `osl`, which was used to solve the set of simultaneous equations in the calibration procedure. Here we show the full module and discuss its structure.

```
"""
A module for one-port vector network analyser (VNA) calibration.

The function :func:`osl` evaluates the three error terms of
a one port: directivity, source match and reflection tracking.

The return type of :func:`osl` is a ``namedtuple``, which
allows the caller to access the error terms by name, e.g.::

    nt = osl(m,n)
    print nt.E_S

The class :class:`OnePort` can be used to store one-port errors
and apply corrections to raw VNA readings.

"""
# BEGIN_preamble
from __future__ import division
from GTC import *

import collections

#-------------------------------------------------------------------------
# A named tuple can be indexed by its element labels
#
OnePortErrors = collections.namedtuple('one_port_errors','E_D E_S E_R')
```

```
# END_preamble

__all__ = (
    'osl',
    'OnePort',
    'OnePortSimpleRandom',
)

# BEGIN_osl
#-----------------------------------------------------------------------
def osl(measured,nominal):
    """
    Return the 1-port errors in a sequence (E_D, E_S, E_R)

    'measured', 'nominal' are 3-element sequences of complex numbers

    The function solves a system of 3 simultaneous equations
    to obtain the three complex reflectometer errors.

    """
    # H =   [ (nominal[0], unity, -nominal[0] * measured[0]),
    #           etc
    #       ]
    H = la.array( [ (n,1.0,-n * m) for m,n in zip(measured,nominal) ] )
    b = la.array( measured )

    ABC = la.solve( H,b )

    E_D=ABC[1]
    E_S=-ABC[2]
    E_R=ABC[0] - ABC[1] * ABC[2]

    return OnePortErrors(E_D,E_S,E_R)
# END_osl

#-----------------------------------------------------------------------
# BEGIN_OnePort
class OnePort(object):

    """
    Corrects raw measurements for systematic one-port errors

    OnePort objects are initialised with the three errors
    normally associated with a VNA port (E_D : directivity,
    E_S : source match, E_R : reflection tracking).

    :method:`correct` is called to correct a raw VNA reading.
    It returns the corresponding error-corrected reflection
    coefficient.

    :method:`correct` calls :method:`random`, which should
    be over-ridden in base classes to include uncertainty
    due to random influences.
    """

    def __init__(self,E_D,E_S,E_R):
        self.E_D = E_D
        self.E_S = E_S
        self.E_R = E_R

    def correct(self,G_raw):
        """
        Return error-corrected reflection coefficient
```

```
        """
        tmp = G_raw - self.E_D
        G = tmp / (self.E_S * tmp + self.E_R)

        return self.random( G )

    # Re-define this in derived classes
    # Here `random` does nothing
    def random(self,g): return g
# END_OnePort
```

Explanation

An extension module is similar to a script. However, the `GTC` modules *must* be imported [2]. This is done by the line `from GTC import *` in the preamble:

```
from __future__ import division
from GTC import *

import collections

#-------------------------------------------------------------------------
# A named tuple can be indexed by its element labels
#
OnePortErrors = collections.namedtuple('one_port_errors','E_D E_S E_R')
```

The line

```
from __future__ import division
```

is also important. This statement must appear at the beginning of the preamble of any extension module. Without it, integer division may result in unwanted truncation. For example,

```
>>> 2 / 4
0
```

but when `__future__.division` is imported, the quotient is a real number

```
>>> 2 / 4
0.5
```

The standard Python module `collections` is imported to get access to the `namedtuple` class, which provides a convenient way of returning several results from a function. In `one_port_cal.py`, the a named tuple `OnePortErrors` can be accessed by the more meaningful attributes: `E_D`, `E_S` and `E_R`. Also, if a `OnePortErrors` object is printed, the attributes are associated with the corresponding elements, which is easier to read.

6.5 Working with Files

6.5.1 Reading and Writing XLS files

[2] When a script is executed, the `GTC` modules are automatically part of the environment. However, this not the case for code that is written inside extension modules.

The XLS file format is used by spreadsheet applications.

GTC includes third-party packages (xlrd, xlwt and xlutils) that support .xls files. These modules can be used to read existing xls files or to create new files.

A good knowledge of Python is required. The package authors have prepared a useful guide, which is available for download (http://www.simplistix.co.uk/presentations/python-excel.pdf).

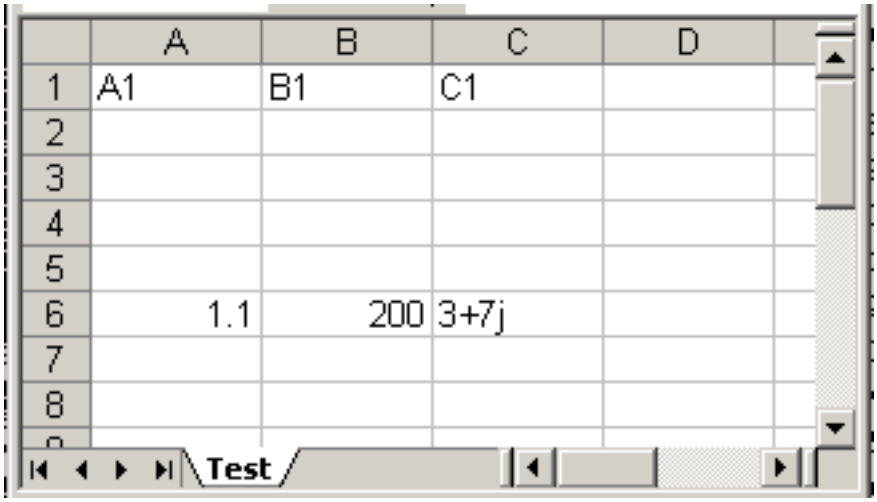
This section gives some very simple examples.

Creating an XLS file

The following code creates a new spreadsheet called demo.xls, with one worksheet Test, and saves it in a folder \tmp.

```
from xlwt import Workbook
from xlwt.Utils import cell_to_rowcol2, col_by_name

#---------------------------------------
def number_to_xls(z):
    """Converts complex to suitable XLS format
    """
    if isinstance(z,complex):
        return ns.to_string(z)
    else:
        return z
#---------------------------------------

# Create an empty workbook
book = Workbook()

# Create a sheet
sheet = book.add_sheet('Test')

# Write to 3 cells on the top row
sheet.write( 0,0, 'A1' )
sheet.write( 0,col_by_name('B'), 'B1' )
r,c = cell_to_rowcol2('C1')
sheet.write( r, c, 'C1' )

# Write to 3 cells on row 6
row = sheet.row(5)
data = (1.1,200,3+7j)
for i,d in enumerate(data):
    row.write(i, number_to_xls(d) )

# Save the file
path = r'C:\tmp\demo.xls'    # Change this to suit
book.save(path)
```

When demo.xls is opened in a spreadsheet application we see



There is a `write` function associated with `sheet` objects that can set cell contents.

The first two arguments to `write` are the row and column indices, respectively. These are integers starting from 0. As an alternative, there is a function `col_by_name` that takes a letter for the column and converts it into an index. There is also a function `cell_to_rowcol2` that converts a normal cell reference (a column letter followed by a row number) into a pair of indices.

The third argument to `write` is the cell value. Strings (first row of the example) and ordinary numbers may be stored directly. However, complex numbers need to be reformatted as strings.

The function `number_to_xls`, defined above, converts a complex number to a string, but leaves all other types of argument unchanged. With this function, all three number types in row 6 of the spreadsheet (float, integer and complex) can be treated the same way.

Reading an XLS file

The following code reads `demo.xls` and prints the numbers

```
from xlrd import open_workbook

# Change this to suit
path = r'C:\tmp\demo.xls'

book = open_workbook( path )
sheet = book.sheet_by_name('Test')

# print row 1
for c in range(3):
    cell = sheet.cell(0,c)
    print cell.value

# Collect row 6 in a list and print it
row6 = [ ns.to_numeric( sheet.cell(5,c).value ) for c in range(3) ]
print "Row 6: %s" % row6
```

The output is

```
A1
B1
C1
Row 6: [1.1, 200.0, (3+7j)]
```

The `cell` function addresses the cells in a `sheet` and the `value` attribute is accesses a cell's contents.

Reading the first row, the column index ranges from 0 to 2 as the strings saved earlier are printed on successive lines.

Reading row 6, a list of numbers is obtained. Again the column index ranges from 0 to 2. Each value is passed to the function *number_strings.to_numeric*, which in this case converts the complex XLS format into a Python complex number.

Writing to an existing XLS file

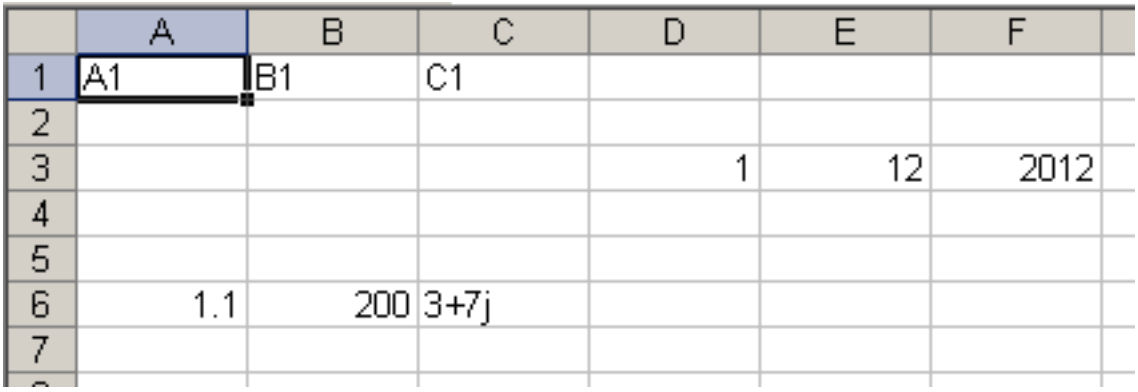
These third-party modules are primarily used for reading an existing file (xlrd) and creating new files (xlwt). However, there may also be a need modify or add data to an existing file.

A function copy allows a workbook that has been opened for reading to be changed into a workbook that can be modified and saved by xlwt. The following short example shows how.

```
from xlrd import open_workbook
from xlutils.copy import copy as copy_workbook

# Change this to suit
path = r'C:\tmp\demo.xls'

#-------------------------------------------------
# Simplify access to worksheets by name
#
def get_sheet(wb,name):
    """Return the worksheet called `name`
    """
    try:
        idx = wb._Workbook__worksheet_idx_from_name[name.lower()]
        return wb._Workbook__worksheets[idx]
    except KeyError:
        print "sheet name %s not found" % repr(name)
#-------------------------------------------------

wb = open_workbook( path )
wb = copy_workbook(wb)

sheet = get_sheet(wb,'Test')

data = (1,12,2012)
r = 2
for c,datum in enumerate(data,3):
    sheet.write(r,c,datum)

wb.save(path) # overwrites
```

The function get_sheet is a convenience, because there is no simple way to obtain a worksheet by name when the workbook object is of the type that can be saved (this is also the case for a new workbook, as in *Creating an XLS file*).

In this example, numbers are added to cells D3, E3 and F3.



	A	B	C	D	E	F
1	A1	B1	C1			
2						
3				1	12	2012
4						
5						
6	1.1	200	3+7j			
7						

Note: When the workbook is saved, the older file is replaced (without warning!).

Warning: Information about macros or user-defined functions is not copied!

6.5.2 Reading and Writing XLSX files

- *Creating an XLSX file*
- *Reading an XLSX file*

The XLSX file is a common in spreadsheet applications.

GTC includes a third-party package (openpyxl) that supports .XLSX files. It can be used to read and create new .xlsx files.

A knowledge of Python is required. The package authors have prepared a useful guide, which is available on-line at http://packages.python.org/openpyxl/tutorial.html.

This section shows some very simple examples.

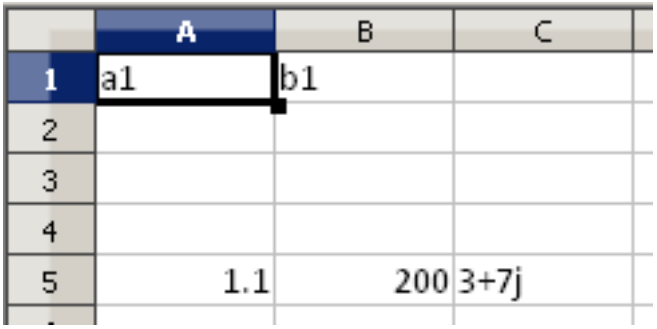
Creating an XLSX file

The following code creates a new spreadsheet called demo.xlsx, with one worksheet Test, and saves it in a folder tmp.

```
import os
import openpyxl

#----------------------------------------
def number_to_xlsx(z):
    """Converts complex to suitable XLSX format
    """
    if isinstance(z,complex):
        return ns.to_string(z)
    else:
        return z

# Change this to suit
path = r'C:\tmp\demo.xlsx'

 # Create an empty workbook
book = openpyxl.Workbook()
```

```
# Get the only sheet
sheet = book.get_active_sheet()
sheet.title = 'Test'

# Write to individual cells
cell_A1 = sheet.cell( 'A1' )
cell_A1.value = 'a1'

# the row and column assignments are needed
# Note that indexing is base-1
cell_B1 = sheet.cell( row=1, column=2 )
cell_B1.value = 'b1'

# Write some numerical data
data = (1.1,200,3+7j)
rng = sheet.range('A6:C6')
for r in rng:
    for cell, d  in zip( r, data ):
        cell.value = number_to_xlsx(d)

book.save(path)
```

When `demo.xlsx` is opened in a spreadsheet application we see



This example shows that the cell contents can be set by assignment to the `value` attribute. Cells can be addressed by the conventional letter-number coordinates, or row and column indices (integers).

Strings and ordinary numbers may be stored directly. However, complex numbers need to be formatted as strings. The function `number_to_xlsx`, defined at the beginning of the example, converts a complex number into a string but leaves all other types of number unchanged.

Reading an XLSX file

The following code reads the file `demo.xlsx` and prints the numbers saved

```
import os
import openpyxl

# Change this to suit
path = r'C:\tmp\demo.xlsx'

book = openpyxl.load_workbook( path )
sheet = book.get_sheet_by_name('Test')

# print row 1
# Note that indexing is base-1
for c in range(1,3+1):
    cell = sheet.cell(row=1,column=c)
    print cell.value
```

```
# Collect row 6 in a list and print it
row6 = [ ns.to_numeric( sheet.cell(row=6,column=c).value ) for c in range(1,3+1) ]
print "Row 6: %s" % row6
```

The output is

```
A1
B1
None
Row 5: [1.1, 200.0, (3+7j)]
```

In this case the `value` attribute of a cell is used to look at the contents.

Reading row 6, a list of numbers is created. Again the column index ranges from 1 to 3. Each value is passed to the function *number_strings.to_numeric*, which converts the complex-number text format back to complex numbers.

6.5.3 Reading and Writing CSV files

The CSV file is used by spreadsheet applications and databases.

A standard Python module can be used to read existing `csv` files or to create new ones.

A good knowledge of Python is required.

This section shows some very simple examples.

Creating a CSV file

The following code creates a file called `demo.csv`, and saves it in a folder `tmp`.

```
import csv

# Labels for columns
columns = ( 'A','B','C')

output_data = [
    (0.01+0.03j,0.02+0.033j,-1),
    (0.97E0-1.397E-7j,0.933333,0.975-0.005j),
    (0.005+0.01j,0.0-0.01j,0.001+0.001j)
]

path = r'C:\tmp\demo.csv'    # Change this to suit
file = open(path,mode='wb') # Must open in 'binary' mode

writer = csv.writer( file )

writer.writerow(columns)
writer.writerows(
    number_strings.sequence_printer(output_data)
)
file.close()
```

When `demo.csv` is opened in a spreadsheet application we see

	A	B	C
1	A	B	C
2	0.01+0.03j	0.02+0.033j	-1
3	0.97-1.397E-07j	0.933333	0.975-0.005j
4	0.005+0.01j	0-0.01j	0.001+0.001j
5			

There is a `writer` class associated with `csv` file objects that can be used to set the contents of cells.

The class has a `writerow` function that takes a sequence of data and writes one row of the file. This is used above to write the three column labels.

The `writer` class also has a `writerows` function that takes a sequence of sequences and stores them as a series of lines. The `output_data` above are stored by this function. Note the use of *`number_strings.sequence_printer`* to convert numbers into a suitable string format before saving them. This ensures that numerical precision is not lost.

Reading an CSV file

The following code reads `demo.csv` and prints the numbers saved earlier.

```
import csv

path = r'C:\tmp\demo.csv'    # Change this to suit

file = open(path,mode='rb') # Open in 'binary' mode
reader = csv.reader( file )

header = reader.next()
table = number_strings.sequence_parser(reader)
file.close()

# Display the data.
#
Nr = len(table)
Nc = len(header)
print  "columns: %s" % str(header)
for r in range(Nr):
    for c in range(Nc):
        print table[r][c],
    print
```

The output is

```
>>> columns: ['A', 'B', 'C']
(0.01+0.03j) (0.02+0.033j) -1
(0.97-1.397e-07j) 0.933333 (0.975-0.005j)
(0.005+0.01j) -0.01j (0.001+0.001j)
```

There is a `reader` class associated with `csv` file objects that can be used to get the contents a file.

The `reader.next()` function returns a sequence corresponding to the next row of data in the file.

The *`number_strings.sequence_parser`* takes a `reader` object and returns a sequence of sequences (rows) of data stored in the file. This quietly converts numbers back from the string format used in the `csv` file.

6.5.4 Archive to a file

The example *Linear Calibration Equations* is used again here to demonstrate archiving.

In this simple scenario, a pressure sensor is calibrated in a laboratory and then returned to the owner with calibration data stored in an archive and a calibration equation defined in GTC code.

There are two parts to the example. First, the coefficients of a calibration line are determined and archived. This is the calibration phase, performed by a calibration laboratory. Then, in a second phase, the sensor is returned to the owner with a report that includes the calibration equation and archived data. This can be used later to convert raw sensor readings into pressures with full uncertainty propagation.

Calibration

The calibration process is carried out

```
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
)

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
```

An archive is created and the results saved

```
sensor_archive = archive.Archive()
sensor_archive.add(a=a,b=b)

file = open('sensor_cal.gar','wb')
archive.dump(file,sensor_archive)
file.close()
```

Using the calibration

Now the sensor has been returned to its owner, with calibration data stored in a file and a calibration equation has been provied in a short GTC script:

```
#--------- Uncertainties
u_lin = type_b.uniform(0.005)
u_res = type_b.uniform(0.00005)

#--------- Calibration equation
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

file = open('sensor_cal.gar','rb')
sensor_archive = archive.load(file)
file.close()
```

This code defines the equation, as well as two sensor uncertainty values (as in *Linear Calibration Equations*). It then loads the archive from the file.

Executing this script with the `GTC` command line option `-i` [1], we may interrogate the `sensor_archive`. There are a number of functions available. For instance,

```
>>> print sensor_archive.keys()
['a', 'b']

>>> print sensor_archive.items()
[('a', ureal(9.0, 0.00015558189517603834, inf)),
    ('b', ureal(5.3895149436550716, 0.0011320432435879308, inf))]
```

If some data is collected, we may apply the calibration function as follows

```
>>> data = [0.4080,0.6120,0.8160,1.0201,1.2242,1.4283]
>>> pressure = [ cal_fn(x) for x in data ]
>>> for p in pressure: print summary(p)
11.1989, u=0.0029, df=inf
12.2984, u=0.0030, df=inf
13.3978, u=0.0030, df=inf
14.4978, u=0.0031, df=inf
15.5978, u=0.0032, df=inf
16.6978, u=0.0033, df=inf
```

We could also manipulate the results of pressure measurements, such as taking the ratio of readings

```
>>> p1_p4 = cal_fn(data[1]) / cal_fn(data[4])
>>> p1_p4
ureal(0.7884668542996432, 0.00023743756188459778, inf)
```

Note that the uncertainty in the ratio is rather smaller than the uncertainty of a single pressure measurement. This is because uncertainty components of the calibration coefficients `a` and `b` are common to both observations [2]. The contribution from these components is nearly canceled in a ratio.

6.5.5 Text File Input and Output

[1] After all script files have been processed, the `-i` option places `GTC` in interactive mode

[2] These common components originate in the pairs of (x,y) data that were used to estimate the slope and intercept of the calibration line.

This section presents two examples that read and write numeric data using text files. A small table of numerical data is stored and retrieved.

- The first example uses a `GTC` class to handle numeric values. Simple Python commands that navigate the Windows file structure are also explained.
- The second example shows how `GTC` functions can convert between numbers and text strings.

Note: Most aspects of reading and writing data from files is described very well by introductory material about Python.

Simple GTC files

This example illustrates how to use *number_strings.File* to work with a file containing numeric data, to make sure that numerical precision is conserved.

The script is included in the `GTC` install directory as `examples\simple_file.py`.

Operating system commands

We first select a file directory using commands from the Python operating system module os.

- `os.getcwd` : returns the current working directory
- `os.chdir` : sets the current working directory
- `os.path.normpath` : normalises the path
- `os.path.exists` : makes sure that the path exists

We display the current directory, prompt for a different path (the current directory is the default) and ensure that any alternative path entered by the user is in fact a valid directory.

```
import os

cwd = os.getcwd()
path = os.path.normpath( raw_input("path [default: %s]: " % cwd) )
assert os.path.exists(path), "'%s' does not exist" % path

os.chdir(path)
print("writing to: %s\\%s" % (os.getcwd(),'data.txt') )
```

Note: the `assert` statement raises an exception if `os.path.exists(path)` is `False`.

Data

A 3-by-3 array of numbers is to be saved. The rows of data are entered in a `list`, where each element is a sequence of three numbers. A sequence of column labels is also defined:

```
output_data = [
    (0.01+0.03j,0.02+0.033j,-1),
    (0.97E0-1.397E-7j,0.933333,0.975-0.005j),
    (0.005+0.01j,0.0-0.01j,0.001+0.001j)
```

```
]
columns = ( 'A','B','C')
```

Writing

We open a file called `data.txt`, write one line containing column labels, then write three more lines of numerical data, separated by commas.

```
with ns.File('data.txt',mode='w',delim=',') as file:
    file.write(columns)
    file.newline()

    # Write each line of data with numbers separated by commas.
    #
    for row in output_data:
        file.write(row)
        file.newline()
```

Reading

This code opens `data.txt`, reads the column headers (first line) and then reads the remaining data.

```
with ns.File('data.txt',mode='r',delim=',') as file:
    cols = file.readline()

    # Read the remaining lines.
    #
    table = file.readlines()
```

The data is displayed by:

```
print( 'columns:'),
for column in header:
    print( column ),
print

for line in table:
    for element in line:
        print( element ),
    print
```

The output will look something like:

```
path [default: C:\Users\user.name\AppData\Local\GTC\examples]
writing to: C:\Users\user.name\AppData\Local\GTC\examples\data.txt
columns: A  B  C
(0.01+0.03j) (0.02+0.033j) -1
(0.97-1.397e-07j) 0.933333 (0.975-0.005j)
(0.005+0.01j) -0.01j (0.001+0.001j)
```

Text files

Standard Python commands are used to read and write strings in a text file. Functions from *number_strings* are used to convert between number and string formats while maintaining numerical precision.

The script is included in the `GTC` install directory as `examples\simple_text_file.py`.

Writing

We use the same *Data* as above. To save this data in a text file, the numbers must be converted to a suitable string format.

A file called `data.txt` is created and the line of column labels is saved, followed by three more lines of numerical data, separated by commas:

```
file = open('data.txt',mode='w')
columns_str = ", ".join(columns)
file.write( columns_str + '\n' )

for row in output_data:
    line_str = ", ".join( number_strings.sequence_printer(row) )
    file.write( line_str + '\n' )

file.close()
```

Note: To form a string by joining the elements of `sequence` we used the Python idiom

```
string = sep.join( sequence )
```

in which the sequence elements will be separated by `sep`. The `GTC` function *`number_strings.sequence_printer`* converts a sequence of numbers into a sequence of strings.

Reading

The file `data.txt` is opened and the column headers (first line) are read. Then the remaining lines that contain the data are read:

```
file = open('data.txt',mode='r')
header = file.readline().strip().split(',')

lines = file.read().splitlines()
file.close()
```

`lines` is a list of strings containing the numbers separated by commas.

The data structure `table` assembled the numeric data. Each row in `table` is a list of numbers converted from the string representation

```
table = []
for s in lines:
    line = s.split(',')
    numbers = number_strings.sequence_parser( line )
    table.append( numbers )
```

Note: `split` breaks a string into a sequence of smaller strings, using the argument as a delimiter. Here, it uses commas to locate the substrings.

`number_strings.sequence_parser` transforms a sequence of strings representing numbers into a sequence of numbers.

The data is displayed by:

```
print( 'columns:'),
for column in header:
```

```
    print( column ),
print

for line in table:
    for element in line:
        print( element ),
    print
```

The output is:

```
path [default: C:\Users\user.name\AppData\Local\GTC\examples]
writing to: C:\Users\user.name\AppData\Local\GTC\examples\data.txt
columns: A  B  C
(0.01+0.03j) (0.02+0.033j) -1
(0.97-1.397e-07j) 0.933333 (0.975-0.005j)
(0.005+0.01j) -0.01j (0.001+0.001j)
```

CHAPTER

SEVEN

FREQUENTLY ASKED QUESTIONS

7.1 What is GTC?

International guidelines on how to calculate measurement uncertainty and report results are readily available [1]. The guidelines are excellent and have been widely adopted.

The GUM Tree Calculator (`GTC`) is a software tool designed to simplify the application of these guidelines. It also extends the approach recommended for real-valued quantities so that problems involving complex-valued quantities can be handled.

`GTC` is developed at the Measurement Standards Laboratory of New Zealand (MSL) [3], the national metrology institute in New Zealand.

`GTC` can be used as an interactive calculator, or as a batch processing tool. It is self-contained (requiring no supporting software), programmable (using the Python language [2]) and can be configured for specific applications.

We think that `GTC` will appeal to a wide range of users, from students of engineering and physical sciences needing to process experimental data, to professional metrologists working in calibration laboratories that have strict requirements for quality standards.

Being a stand-alone application, `GTC` is resilient to changes elsewhere in a computer system. This is important in the context of quality standards such as ISO 17025 [4] (which many calibration laboratories adhere to, including

[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

[3] http://msl.irl.cri.nz/

[2] http://www.python.org/

[4] ISO/IEC 17025:2005, *General requirements for the competence of testing and calibration laboratories* (see http://en.wikipedia.org/wiki/ISO/IEC_17025)

MSL).

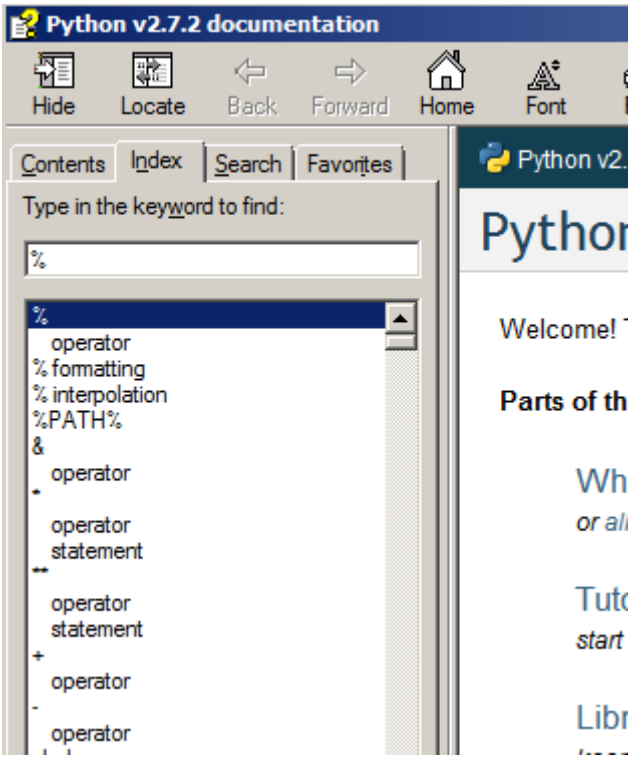
GTC runs on Windows operating systems [5]. It is a simple console application (i.e. command prompt) without a graphical user interface.

7.2 What does that funny symbol mean?

Python defines symbols for a variety of operations. For example, ** represents 'to the power of', e.g.

```
>>> 2**3
8
```

If you encounter a symbol and want to find out what it means, use the Python Help file. Type in the symbol at the index search tab on the left. Here is an example using the % symbol



The information available may be a little terse. If so, choose a few key words from the Help file and then do an internet search on them.

For example, suppose we need information about %. We select the % formatting entry in the Help, which is the right topic, but find the information there difficult to follow. Nevertheless, from what we see, the topic of interest is something like 'string formatting'. So, then an internet search on "Python string formatting tutorial" brings up a range of helpful material.

7.3 How do I report a bug in GTC?

Please send us a **short** example that demonstrates the problem.

[5] GTC is a 'win32 console' application. To date, it has been used successfully with Windows XP, Windows Vista, Windows 7 and Windows 10.

7.4 Can I do a type-A analysis on a set of uncertain numbers?

There will be situations where data is subject to some errors that are accounted for by type-B uncertainty components, while other sources of error cause the observations to fluctuate, so a type-A analysis would be appropriate.

For example, an instrument readout is subject to an unknown offset error, due to imperfect adjustment, in addition to noise that changes from one reading to the next. The offset error estimate is zero, but there is uncertainty associated with this estimate.

Suppose five readings are observed when the input stimulus is fixed

```
1.2090047238650097, 1.1920239005254694, 1.2023705451450373,
1.1999502398020352, 1.2120802857623008
```

and the type-B standard uncertainty associated with the offset error is 0.01.

We first create a sequence of uncertain-number data in which each element is influenced by the *same* offset error

```
>>> x = [1.2090047238650097, 1.1920239005254694, 1.2023705451450373,
... 1.1999502398020352, 1.2120802857623008]
>>> e_offset = ureal(0,0.01,label='offset')
>>> data = [ x_i + e_offset for x_i in x ]
```

A type-A analysis of this data can be done using *estimate*, which looks only at the values of the elements in `data`

```
>>> data_a = type_a.estimate( data, label='type-A' )
>>> data_a
ureal(1.2030859390199704, 0.003525928111849485, 4, label=type-A)
```

This is a conventional type-A evaluation of the sample mean, based on the five observations.

Independently, we may do a calculation **with** propagation of uncertainty using *function.mean*

```
>>> data_b = function.mean(data)
>>> data_b
ureal(1.2030859390199704, 0.010000000000000002, inf)
```

In that case the uncertainty associated with the mean is the same as the uncertainty of a single reading, because the adjustment error is systematic and is not reduced by averaging.

In one last step, these two results can be combined to obtain an uncertain number that allows for both the influence of random noise and the offset error

```
>>> mu = type_a.merge_components(data_a,data_b)
>>> mu
ureal(1.2030859390199704, 0.01060340365401273, 327.15032481132585)
>>> for cpt in rp.budget(mu,trim=0):
...         print "  %s: %G" % cpt
...
  offset: 0.01
  type-A: 0.00352593
```

Note: Several function names are deliberately paired in the *type_a* and *function* modules, to suggest that this type of analysis can be carried out. Various types of linear regression may be used in this way, as well as the calculation of a mean, as shown here.

7.5 Can I use CSV (comma-separated value) files?

It is possible to read and write comma-separated value (CSV) files. There is a standard Python module available for this.

Some knowledge of Python is required.

A simple example is given in *Reading and Writing CSV files*.

7.6 Can I use `.xls` spreadsheet files?

Three third-party packages (`xlrd`, `xlwt` and `xlutils`) that support `.xls` files are included in the installation directory (usually `C:\Users\user.name\AppData\Local\GTC\lib`). With these modules it is possible to read and write `.xls` files.

A useful guide is available http://www.simplistix.co.uk/presentations/python-excel.pdf.

Some knowledge of Python is required.

A simple example with `GTC` is given in *Reading and Writing XLS files*.

7.7 Can I use `.xlsx` spreadsheet files?

A third-party package (`openpyxl`) that supports `.xlsx` files is included in `GTC`. With this module it is possible to read and write `.xlsx` files, although not while they are open in a spreadsheet application.

A useful guide is available at http://packages.python.org/openpyxl/tutorial.html.

Some knowledge of Python is required.

A simple example is given in *Reading and Writing XLSX files*.

7.8 Can I use RTF (rich text format) files?

A third-party package called `PyRTF` is included in the installation directory (usually `C:\Users\user.name\AppData\Local\GTC\lib`).

See http://pyrtf.sourceforge.net/ for further details.

7.9 How do I define an uncertain number with relative uncertainty?

Suppose that measurement of a quantity A yields $a = 125.56$ with a relative uncertainty of $u(a)/a = 0.05$. We can define an uncertain number `e_rel` for the relative error

```
>>> a = 125.56
>>> e_rel = ureal(1,0.05)
```

then an uncertain number representing A is

```
>>> A = a * e_rel
>>> A
ureal(125.56, 6.2780000000000005, inf)
```

7.10 Is there a simple way to chain GTC calculations?

Often a calculation proceeds in distinct stages and it is convenient to separate these in different files.

If the first part of a calculation is contained in one file but the remaining steps are in another. The results of the first calculation can be passed to the second.

Multiple files can be processed in GTC by putting their names together on the command line. For example,

```
C:\my work>gtc first_file.py second_file.py
```

would process first_file.py and then second_file.py, as if they were just one big file.

If this method is not suitable, then *Storing uncertain numbers* and restoring them in a later calculation is another possibility.

7.11 Why does the GTC window close before I can read anything?

If an error occurs running GTC, the GTC command window closes before the error message can be read.

Try running the same script using *The SciTE editor*, which will show error messages in the output window. Alternatively, open a Command Prompt window and then run the script (see *The Command Prompt*). The Command Prompt window should stay open and display any error messages.

Part IV

Reference

CHAPTER

EIGHT

GTC MODULES

The functions and classes that make up GTC.

8.1 Core Functions and Classes

The functions and classes defined in the *core* module are automatically available.

Note: In user-defined extension modules, the GTC modules must be imported. For example, `from core import *` must added at the top of extension module files that use core functions or classes.

8.1.1 Uncertain Number Types

There are two types of uncertain number, one for real quantities and one for complex quantities.

Uncertain Real Numbers

The class *UncertainReal* represents uncertain real numbers.

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) are applied to the *value* of an uncertain number. For example,

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

An *UncertainReal* may be converted to a real number using the Python function `float`.

```
>>> un = ureal(1,1)
>>> math.asin( float(un) )
1.5707963267948966
```

When an *UncertainReal* is converted to a string (e.g., by the `str` function, or by using the `%s` string-conversion format specifier), the value is embedded between `?`'s. The precision depends on the uncertainty. For example,

```
>>> x = ureal(1.414141,0.01)
>>> print str(x)
?1.41?
>>> print "%s" % x
?1.41?
```

When an *UncertainReal* is converted to its Python *representation* (e.g., by `repr` or by using the `%r` string-conversion format specifier), a string is returned showing the full internal precision of numbers. For example,

```
>>> x = ureal(1.414141,0.01,5,label='x')
>>> print repr(x)
ureal(1.4141410000000001, 0.01, 5, label=x)
>>> print "%r" % x
ureal(1.4141410000000001, 0.01, 5, label=x)
```

UncertainReal objects have the attributes `x`, `u`, `v`, `df`, and `s`, which obtain the value, uncertainty, variance, degrees-of-freedom and a summary string for the uncertain number, respectively.

The documentation for *UncertainReal* follows.

class **UncertainReal** (*x*, *u_comp*, *i_comp*, *node*, *c*)

The class implementing uncertain real numbers

df

The degrees of freedom attribute

Returns float

Example::

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

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

label

The label attribute

A `label` may be set by assignment.

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

Example::

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

```
>>> y = 5 * x
>>> y.label = 'y'
>>> y.label
'y'
```

```
>>> label(y)
'y'
```

s

A summary string

The format consists of the label (if defined) and three numbers: the value, standard uncertainty and degrees-of-freedom.

The uncertainty is reported to two significant figures and the value uses the same precision. The degrees-of-freedom are rounded down to the nearest integer.

Returns string

Example::

```
>>> ur = ureal(2.5,0.5,3,label='x')
>>> ur.s
'x:   2.50, u=0.50, df=3'
```

```
>>> ur = ureal(2.5,0.5,3)
>>> ur.s
' 2.50, u=0.50, df=3'
```

Note: `un.s` is equivalent to `summary(un)`

`df=nan` indicates that the degrees-of-freedom calculation is invalid.

`df=inf` indicates that the of degrees-of-freedom is greater than 1E6.

---

u

The standard uncertainty attribute

Returns float

Example::

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

Note: `un.u` is equivalent to `uncertainty(un)`

---

v

The standard variance attribute

Returns float

Example::

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

Note: `un.v` is equivalent to `variance(un)`

x

The value attribute

Returns float

Example::

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

Note: `un.x` is equivalent to `value(un)`

Uncertain Complex Numbers

The class *UncertainComplex* represents uncertain complex numbers.

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

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

Equality comparison operations (== and !=) are applied to the *value* of uncertain complex numbers. For example,

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

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

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

>>> magnitude(uc)
ureal(1.4142135623730951, 0.99999999999999989, inf)
```

An `UncertainComplex` may be converted to a complex number using the Python function `complex`.

```
>>> uc = ucomplex(4j,(1,1))
>>> math.sqrt( complex(uc) )
(1.4142135623730951+1.4142135623730949j)
```

When an *UncertainComplex* is converted to a string (e.g., by the `str` function or by using the `%s` string-conversion format specifier), the value is embedded between `?`'s. The precision depends on the uncertainty. For example,

```
>>> z = ucomplex(1.333-0.121212j,(0.01,0.01))
>>> print( str(z) )
?(1.33-0.12j)?
>>> print( "%s" % z )
?(1.33-0.12j)?
```

When an *UncertainComplex* is converted to its Python *representation* (e.g., by `repr` or by using the `%r` string-conversion format specifier), a string is returned in which the full internal precision of numbers is displayed. For example,

```
>>> z = ucomplex(1.333-0.121212j,(0.01,0.01),7,label='z')
>>> print( repr(z) )
ucomplex( (1.333-0.121212j), [0.0001,0.0,0.0,0.0001], 7, label=z )
>>> print( "%r" % z )
ucomplex( (1.333-0.121212j), [0.0001,0.0,0.0,0.0001], 7, label=z )
```

Note that the variance-covariance matrix is shown as a 4-element list.

UncertainComplex objects have attributes `x`, `u`, `v`, `df`, and `s`, which obtain the value, uncertainty, variance-covariance matrix, degrees-of-freedom and summary string of the uncertain number, respectively.

The documentation of *UncertainComplex* follows.

class UncertainComplex (*r*, *i*)

A class representing uncertain complex numbers

df

The degrees-of-freedom attribute

When the object is not an elementary uncertain number, the effective degrees-of-freedom is calculated by the function `willink_hall`.

Returns float

Example::

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

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

label

The *label* attribute

A `label` may be set by assignment.

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

Example::

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

```
>>> zz = z * z.conjugate()
>>> zz.label = 'zz'
```

```
>>> label(zz)
'zz'
```

s

A summary string

The summary string is composed of the label (when defined), the value, the standard uncertainties, the correlation coefficient and the degrees-of-freedom.

Returns string

Example ::

```
>>> uc = ucomplex(1+2j,(.3,.2),3,label='x')
>>> uc.s
'x: (1.00,2.00), u=[0.30,0.20], r=0.00, df=3'
```

```
>>> uc = ucomplex(1+2j,(.3,.2),3)
>>> uc.s
'(1.00,2.00), u=[0.30,0.20], r=0.00, df=3'
```

Note: `uc.s` is equivalent to `summary(uc)`

The calculation of degrees-of-freedom is invalid if `df=nan`.

When `df=inf` is indicated, the of degrees-of-freedom is greater than 1E6.

---

u

The standard uncertainty attribute (real and imaginary components)

Returns 2-element sequence of float

Example::

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

Note: `uc.u` is equivalent to `uncertainty(uc)`

---

v

The variance-covariance attribute

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

Returns 4-element sequence of float

Example::

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

Note: `uc.v` is equivalent to `variance(uc)`

---

x

The value attribute

Returns complex

Example::

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

Note: `uc.x` is equivalent to `value(uc)`

8.1.2 Core Functions

A set of mathematical functions is defined in the *core* module, together with functions that create elementary uncertain numbers and functions that access uncertain number attributes.

Basic functions for uncertainty calculations

Creation of uncertain numbers

- *ureal* : creates an elementary uncertain real number
- *ucomplex* : creates an elementary uncertain complex number
- *result* : creates an intermediate uncertain number
- *constant* : creates a constant uncertain number (with no uncertainty)
- *multiple_ureal* : creates a set of related elementary uncertain real numbers
- *multiple_ucomplex* : creates a set of related elementary uncertain complex numbers

Uncertain number attributes

- *value* : the value
- *uncertainty* : the standard uncertainty
- *variance* : the standard variance
- *dof* : the degrees-of-freedom
- *label* : the label
- *summary* : the summary string

Relationships between uncertain numbers

- *component* : a component of uncertainty
- *set_correlation* : assign a correlation coefficient
- *get_correlation* : evaluate a correlation coefficient
- *get_covariance* : evaluate covariance

Uncertain number math functions

- *cos*
- *sin*
- *tan*
- *acos*
- *asin*
- *atan*
- *atan2*
- *exp*
- *log*
- *log10*
- *sqrt*
- *sinh*
- *cosh*
- *tanh*
- *asinh*
- *acosh*
- *atanh*
- *mag_squared*
- *magnitude*
- *phase*

ureal (*x*, *u*, *df=inf*, *label=None*, *dependent=False*)
Create an elementary uncertain real number

Parameters

- **x** (*float*) – the value (estimate)
- **u** (*float*) – the standard uncertainty
- **df** (*float*) – the degrees-of-freedom
- **label** (*string*) – a string label
- **dependent** (*Boolean*) – allows setting a correlation coefficient

Return type UncertainReal

Example:

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

multiple_ureal (*x_seq*, *u_seq*, *df*, *label_seq=None*)
Return a sequence of related elementary uncertain real numbers

Parameters

- **x_seq** – a sequence of values (estimates)
- **u_seq** – a sequence of standard uncertainties

- **df** – the degrees-of-freedom
- **label_seq** – a sequence of labels

Return type a sequence of `UncertainReal`

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

Correlation between any pairs of the uncertain numbers defined does 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)
>>> summary(r)
'127.732, u=0.070, df=4'
>>> reporting.uncertainty_interval(r)
expanded_uncertainty(lower=127.53788813535775, upper=127.92645172084642)
```

ucomplex(*z*, *u*, *df=inf*, *label=None*, *dependent=False*)

Create an elementary uncertain complex number

Parameters

- **z** (*complex*) – the value (estimate)
- **u** (*a float, 2-element or 4-element sequence*) – the standard uncertainty or variance
- **df** (*float*) – the degrees-of-freedom
- **label** (*string*) – a string label
- **dependent** (*Boolean*) – allows setting a correlation coefficient

Return type `UncertainComplex`

`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 equal to `u`.

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

If `u` is a 4-element sequence, it is associated with a variance-covariance matrix.

A *RuntimeError* is raised if `df` or `u` have illegal values.

Examples:

```
>>> uc = ucomplex(1+2j,(.5,.5),3,label='x')
>>> uc
ucomplex((1+2j), u=[0.5,0.5], r=0, df=3, label='x')
```

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

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 (estimates)
- **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 related uncertain complex numbers with the same number of degrees-of-freedom.

Correlation between pairs of these uncertain numbers does 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
>>> z
ucomplex(
    (127.73216992810208+219.84651191263839j),
    u=[0.069978727988371722,0.29571682684612355],
    r=-0.59099999999999997,
    df=5
)
>>> reporting.uncertainty_interval(z.real)
expanded_uncertainty(lower=127.55228662093492, upper=127.91205323526924)
```

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

Declare `un` as an intermediate uncertain number 'result'

un - an uncertain number *label* - a label can be assigned

The dependence of other uncertain numbers on this result can be stored in an archive.

Call this function other uncertain numbers are created that depend on it.

Example:

```
# The dependence of `P` on `V`
# can be archived

V = result( I*R )
P = V**2/R
```

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(2.71828,0,inf,label='Euler')
```

value (*x*)
Return the value

Returns a complex number if x is an uncertain complex number

Returns a real number if x is an uncertain real number

Returns x otherwise.

Example:

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

uncertainty (*x*)
Return the standard uncertainty

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

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

Otherwise, return 0.

Examples:

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

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

variance (*x*)
Return the standard variance

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

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

Otherwise, return 0.

Examples:

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

>>> uc = ucomplex(1+2j,(.5,.5),3,label='x')
```

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

dof (*x*)

Return the degrees-of-freedom

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

Returns `nan` when the calculation is invalid

Note: This function calls *reporting.variance_and_dof*, which evaluates both the variance and the degrees-of-freedom. It may be more efficient to call that function directly in some situations.

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
```

label (*x*)

Return the label

Returns `None` when no label has been assigned

Example:

```
>>> x1 = ureal(2,1,label='x1')
>>> label(x1)
'x1'
>>> x1.label
'x1'
```

summary (*x*)

Return a summary string

The string format is

```
label : value, uncertainty, dof
```

or, if no label is defined,

```
value, uncertainty, dof
```

The precision used to display the value depends on the uncertainty

For an uncertain real, the standard uncertainty is shown.

For an uncertain complex, a pair of standard uncertainties for the real and imaginary components is shown.

When `df=inf` the degrees of freedom is greater than 1E6.

If `x` is not an uncertain number, the Python representation is returned.

Examples:

```
>>> x1 = ureal(2.1213,0.5,label='x1')
>>> summary(x1)
'x1: 2.12, u=0.50, df=inf'
```

```
>>> z1 = ucomplex(1.24242-3.14141j,(0.1,0.1),5,label='z1')
>>> summary(z1)
'z1: (1.24-3.14j), u=[0.10,0.10], r=0.00, df=5'
```

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 magnitude of *reporting.u_component* is returned.

If either `x` or `y` is uncertain complex, the returned value represents the magnitude of the component of uncertainty matrix (see also: *reporting.u_component* and *reporting.u_bar*).

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

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)
u_components(rr=-1.0, ri=0.0, ir=0.0, ii=-1.0)
>>> component(y,z2)
1.0
```

get_covariance (*arg1*, *arg2=None*)

Return the covariance

The arguments `arg1` and `arg2` may be:

•a pair of uncertain real numbers,

•a single uncertain complex number

•a pair of uncertain complex numbers.

The return value may be a real number or a sequence of four real numbers.

When a pair of uncertain real numbers are given, return the covariance between them.

When a single uncertain complex number is given, return the covariance between the real and imaginary components.

When there are two arguments and at least one is a complex number, or an uncertain complex number, a 4-element sequence is returned.

When two uncertain complex arguments are given, the 4-element sequence representing the covariance between the real and imaginary components is returned.

If a numerical argument is used, zero, or a 4-element sequence of zeros, is returned.

Raises `RuntimeError` when illegal arguments types are used

Example:

```
>>> x1 = ureal(2,0.5)
>>> x2 = ureal(5,2.1)
>>> x3 = ureal(5,0.75)
>>> x4 = x1 + x2
>>> x5 = x2 + x3
>>> get_covariance(x4,x5)
4.41

>>> x1 = ucomplex(1,(0.5,1.2),dependent=True)
>>> x2 = ucomplex(1,(1.3,1.9),dependent=True)
>>> correlation_mat = (0.25,0.5,0.75,0.5)
>>> set_correlation(correlation_mat,x1,x2)
>>> get_covariance(x1,x2)
covariance_matrix(rr=0.1625, ri=0.475, ir=1.17, ii=1.14)
```

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

Return the correlation coefficient

`arg1` and `arg2` may be:

- a pair of uncertain real numbers,
- a single uncertain complex number
- a pair of uncertain complex numbers.

The return value may be a real number or a sequence of four real numbers.

When a pair of uncertain real numbers are used, return the correlation between them.

When a single uncertain complex number is used, return the correlation between the real and imaginary components.

When there are two arguments and at least one is a complex number, or an uncertain complex number, return a 4-element sequence.

When two uncertain complex number are given, a 4-element sequence representing the correlation coefficients between the real and imaginary components is returned.

If a number is given as an argument, zero, or a sequence of zeros, is returned.

Raises `RuntimeError` when illegal arguments types are used

Example:

```
>>> x1 = ureal(2,1)
>>> x2 = ureal(5,1)
>>> x3 = ureal(5,1)
>>> x4 = x1 + x2
>>> x5 = x2 + x3
>>> get_correlation(x4,x5)
0.5

>>> x1 = ucomplex(1,(1,1),dependent=True)
>>> x2 = ucomplex(1,(1,1),dependent=True)
>>> correlation_mat = (0.25,0.5,0.75,0.5)
>>> set_correlation(correlation_mat,x1,x2)
>>> get_correlation(x1,x2)
correlation_matrix(rr=0.25, ri=0.5, ir=0.75, ii=0.5)
```

set_correlation(*r*, *arg1*, *arg2=None*)

Set the correlation between elementary uncertain numbers

The input arguments may 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.

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

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

`RuntimeError` is raised when illegal arguments are used

A warning will be issued if the uncertain number arguments have not been declared as `dependent`.

Examples:

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

>>> z = ucomplex(1+0j,(1,1),dependent=True)
>>> z
ucomplex((1+0j), u=[1,1], r=0, df=inf)
>>> set_correlation(0.5,z)
>>> z
ucomplex((1+0j), u=[1,1], r=0.5, df=inf)

>>> x1 = ucomplex(1,(1,1),dependent=True)
>>> x2 = ucomplex(1,(1,1),dependent=True)
>>> correlation_mat = (0.25,0.5,0.75,0.5)
>>> set_correlation(correlation_mat,x1,x2)
>>> get_correlation(x1,x2)
correlation_matrix(rr=0.25, ri=0.5, ir=0.75, ii=0.5)
```

cos (*x*)
Uncertain number cosine function

sin (*x*)
Uncertain number sine function

tan (*x*)
Uncertain number tangent function

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

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

---

asin (*x*)
Uncertain number arcsine function

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

---

atan (*x*)
Uncertain number arctangent function

Note: In the complex case there are two branch cuts: One extends from j along the imaginary axis to j∞, 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
ureal(0.523598775598299,1,inf)
>>> math.degrees( theta.x )
30.000000000000004
```

exp (*x*)
Uncertain number exponential function

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

Raises x to the power of y

log (*x*)
Uncertain number natural logarithm

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

---

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

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

---

sqrt (*x*)
Uncertain number square root function

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

---

sinh (*x*)
Uncertain number hyperbolic sine function

cosh (*x*)
Uncertain number hyperbolic cosine function

tanh (*x*)
Uncertain number hyperbolic tangent function

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

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

asinh(*x*)
Uncertain number hyperbolic arcsine function

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

atanh(*x*)
Uncertain number hyperbolic arctangent function

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

phase(*z*)
Return an uncertain real number for the phase

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

Returns the phase in radians

Return type `UncertainReal`

8.2 Evaluating type-A uncertainty

A type-A evaluation of uncertainty involves statistical analysis of data. In contrast, type-B uncertainty is obtained without statistical analysis.

The prefix *type_a* (or the alias `ta`) is needed as to resolve the names of objects defined in this module.

8.2.1 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.

8.2.2 Correcting indications

- *BiasedIndication* is a class of objects that can be used to correct future indications for bias using a type-A estimate of required the correction term.

8.2.3 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.
- *line_fit_rwls* performs a weighted least-squares straight line fit to a sample of data. In this case, the weights are used to normalise the variability of observations.
- *line_fit_wtls* performs a weighted total least-squares straight line fit to a sample of data.
- *chisq_p* and *chisq_q* evaluate the Chi-square probability functions `P(nu,x)` and `Q(nu,x) = 1 -P(nu,x)`.

8.2.4 Merging uncertain components

- *merge_components* combines the results from type-A and type-B analyses.

Note: Many functions in *type_a* treat the data as pure numbers. Sequences of uncertain numbers can be passed to these functions, but only the values of the uncertain numbers will be used. This allows type-B uncertainty components to be associated with observational data (e.g., the type-B uncertainty due to a systematic error) before a type-A analysis is performed, which is often convenient.

merge_components is provided so that the results of type-A and type-B analyses on the same data sequence can be combined. Note, however, that doing so may over-emphasize uncertainty components that contribute to variability in the observations.

8.2.5 Module contents

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

Obtain an uncertain number by type-A evaluation

Parameters

- **seq** – a sequence representing a sample of data
- **label** – a label for the returned uncertain number

Returns an uncertain real number, or an uncertain complex number

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

The sample mean is 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, for the complex case).

Returns an uncertain real number when the mean of `seq` is real, or an uncertain complex number when the mean is complex.

Examples:

```
>>> data = range(15)
>>> type_a.estimate(data)
ureal(7,1.15470053837925,14)

>>> 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+0.9574410497332931j),
    u=[0.2888166531024181,0.2655555630050262],
    r=-0.314,
    df=9
)
```

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

Return a sequence of related uncertain real numbers

Parameters

- **seq_of_seq** – a sequence of real-valued sequences
- **labels** – a sequence of labels

Returns a sequence of uncertain real numbers

The sequences in `seq_of_seq` must all be the same length.

Defines uncertain numbers using the sample statistics from the data sequences, including the sample covariance.

The uncertain numbers returned are considered related, so that a degrees-of-freedom calculation can be performed even if there is correlation between them.

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(4.9989999999999967,0.0032093613071761794,4,label='V')
>>> i
ureal(0.019661000000000001392,9.4710083940413345689e-06,4,label='I')
>>> p
ureal(1.044459999999999944,0.00075206382707853681 49,4,label='phi')

>>> r = v/i*cos(p)
>>> r
ureal(127.73216992810208,0.071071407396995398,4)
```

multi_estimate_complex(*seq_of_seq*, *labels=None*)
Return a sequence of related uncertain complex numbers

Parameters

- **seq_of_seq** – a sequence of complex number sequences
- **labels** – a sequence of labels for the uncertain numbers

Returns a sequence of uncertain complex numbers

The sequences in `seq_of_seq` must all be the same length.

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

The uncertain complex numbers returned are considered related, so they may be used in a degrees-of-freedom calculation even if there is correlation between them.

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(127.73216992810208,0.071071407396995398,4)
>>> get_correlation(z.real,z.imag)
-0.5884297844235157
```

estimate_digitized(*seq*, *delta*, *label=None*, *truncate=False*)
Return an uncertain number for the mean of a sample of digitized data

Parameters

- **seq** – a sequence of real numbers or uncertain real numbers
- **delta** – a real number for the digitization step size
- **label** – a label for the returned uncertain number
- **truncate** – if True, truncation is assumed

When an instrument rounds, or truncates, readings to a finite resolution `delta`, the uncertainty in an estimate of the mean of a sequence of readings depends on the amount of scatter in the data and on the number of points in the sample.

The argument `truncate` should be set `True` if an instrument truncates readings instead of rounding them.

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(-0.005627272727272727874,1.9497827808661157478e-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(-0.005636363636363636364021,1.5212000482437778871e-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(-0.0055999999999999999431,2.8867513459481289171e-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(-0.0055999999999999999431,3.2914029430219170322e-05,2)
```

mean(*seq*)

Return the arithmetic mean of data in `seq`

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

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

Example:

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

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

Return the sample standard deviation

Parameters

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

If `seq` contains complex or uncertain complex numbers, the standard deviation in the real and imaginary components is evaluated, as well as the sample correlation coefficient.

Otherwise the sample standard deviation is returned.

The calculation only uses the *value* attribute of uncertain numbers.

Examples:

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

>>> data = [(0.91518731126816899+1.52134429555755518j),
... (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
standard_deviation(real=0.913318449990377, imag=0.8397604244242309)
>>> r
-0.31374045124595246
```

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

Return the standard uncertainty of the sample mean

Parameters

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

Return type float

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.

Otherwise the standard uncertainty of the sample mean is returned.

The calculation only uses the *value* attribute of uncertain numbers.

Example:

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

>>> data = [(0.91518731126816899+1.52134429555755518j),
... (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
standard_uncertainty(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 numbers
- **mu** – the arithmetic mean of `seq`

Returns a 4-element sequence

If `mu` is not provided it will be evaluated (see *mean*).

`seq` may contain numbers or uncertain numbers. However, the calculation only uses the *value* of uncertain numbers.

Example:

```
>>> data = [(0.91518731126816899+1.52134429555755518j),
... (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)
variance_covariance(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
```

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

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

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(4.81388888888888,4.88620631218336,7)
>>> b
ureal(9.408333333333335,0.868301647656361,7)

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

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

Return a weighted least-squares straight-line fit

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
 intercept_slope(
    a=ureal(0.8852320675105488,0.5297081435088364,inf),
    b=ureal(2.056962025316456,0.177892016741205,inf)
)
```

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

Return a relative weighted least-squares straight-line fit

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:

Number of points: 6
Intercept: 1.14, u=0.12, df=4
Slope: 1.973, u=0.041, df=4
Correlation: -0.87
Sum of the squared residuals: 1.33952
```

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

Return a total least-squares straight-line fit

Parameters

- **a0_b0** – initial line intercept and slope
- **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`
- **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*

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]

# initial estimates are needed
>>> a0_b0 = type_a.line_fit(x,y).a_b

# 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 = type_a.line_fit_wtls(a0_b0,x,y,ux,uy)
>>> result.a_b
intercept_slope(
    a=ureal(5.479910183283027,0.2919334989452106,8),
    b=ureal(-0.48053339910867754,0.057616740759399841,8)
)
```

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

This object holds the results of an ordinary least-squares regression to data.

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

N

The number of points in the sample

a_b

Return the intercept and slope as uncertain numbers

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*, *label=None*)

Estimate the stimulus `x` that caused the response `yseq`.

Parameters

- **yseq** – a sequence of further observations of `y`
- **label** – a label for the estimate of *y* based on `yseq`

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] )
>>> summary(x0)
'0.260, u=0.018, df=13'
```

y_from_x (*x*, *label=None*)
Return an uncertain number y for the response to x

Parameters **x** – a real number, or an uncertain real number

Estimates the response y that might be observed for a stimulus x

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

class LineFitWLS (*a*, *b*, *ssr*, *N*)
This object holds results from a weighted LS linear regression to data.

N
The number of points in the sample

a_b
Return the intercept and slope as uncertain numbers

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.

class LineFitRWLS (*a*, *b*, *ssr*, *N*)
This object holds the results of a relative weighted least-squares regression. The weight factors provided normalise the variability of observations.

N
The number of points in the sample

a_b
Return the intercept and slope as uncertain numbers

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*, *label=None*)
Estimates the stimulus x that generated the response sequence yseq

Parameters

- **yseq** – a sequence of further observations of y
- **s_y** – a scale factor for the uncertainty of the yseq
- **label** – a label for the estimate of *y* based on yseq

y_from_x (*x*, *s_y*, *label=None*)
Return an uncertain number y for the response to x

Parameters

- **x** – a real number, or an uncertain real number
- **s_y** – a scale factor for the response uncertainty

Estimates the response y that might be generated by 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.

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

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

N

The number of points in the sample

a_b

Return the intercept and slope as uncertain numbers

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.

merge_components(*a*, *b*)

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

Parameters

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

Returns an uncertain number that combines the uncertainty components of `a` and `b`

The values of `a` and `b` must be equal 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.

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
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 = function.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_components(y_1_a,y_1_b)
y_2 = type_a.merge_components(y_2_a,y_2_b)
```

chisq_p(*nu*, *x*)

Return the propability that chi-squared could be less than `x`

Parameters

- **nu** – the number of degrees of freedom
- **x** – sum of squared residuals

Returns the incomplete gamma function `P(nu,x)`

`P(nu,x)` is the probability that any random set of *N* points would give a value of chi-squared less than the observed value `x`

See: P R Bevington, *Data reduction and error analysis for the physical sciences* (McGraw-Hill)

Example:

```
>>> type_a.chisq_p(10,3.94)
0.049986909209909315
```

chisq_q(*nu*, *x*)

Return the propability that chi-squared could exceed `x`

Parameters

- **nu** – the number of degrees of freedom
- **x** – sum of squared residuals

Returns the incomplete gamma function `Q(nu,x) = 1 -P(nu,x)`

`Q(nu,x) = 1 -P(nu,x)` is the probability that any random set of *N* points would give a value of chi-squared greater than or equal to the observed value `x`

See: P R Bevington, *Data reduction and error analysis for the physical sciences* (McGraw-Hill)

Example:

```
>>> type_a.chisq_q(10,3.94)
0.9500130907900907
```

This function might be used after a weighted least-squares regression has been performed, to assess the goodness of fit. *chisq_q* can calculate the probability that a value of sum-squared-residuals could be exceeded by chance. (See *Numerical Recipes in C: The Art of Scientific Computing*, Section 15.1).

class **BiasedIndication**(*correction*)

An object to correct single observations using a type-A estimate of bias. The corrected result is an uncertain number with uncertainty components that depend on the variability of observations and the uncertainty of the bias estimate.

In statistical parlance, the uncertain numbers produced by this object can be used to calculate a prediction interval for future observations.

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

Return an uncertain number for the offset-corrected indication

Parameters

- **x** – a real number
- **label** – a label for the indication uncertainty

Returns an uncertain number

The uncertainty of a corrected indication has a component due to the reading variability and a component due to the uncertainty of the estimated additive correction.

In statistical terminology, the uncertain number can be used to calculate a 'prediction interval' for a future indication.

Example:

```
# `sample` is a sequence of `N` indications
# First estimate the bias (offset)
x_bar = type_a.estimate(sample)

# Then create an object to process
# other indications
processor = type_a.BiasedIndication(x_bar)

# x_i is another indication and
# x_corr_i is an uncertain number for
# the offset-corrected indication.
x_corr_i = processor.offset(x_i)
```

8.3 Evaluating type-B uncertainty

The prefix *type_b* (or the alias `tb`) is needed as to resolve the names of objects defined in this module.

8.3.1 Real-valued problems

The following functions convert the half-width of a one-dimensional distribution to a standard uncertainty:

- *uniform*
- *triangular*
- *u_shaped*
- *arcsine*

8.3.2 Complex-valued problems

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

- *uniform_ring*
- *uniform_disk*
- *unknown_phase_product*

8.3.3 A table of distributions

The mapping `distribution` allows the functions above to be selected by name. For example,

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

```
>>> ureal( 1, type_b.distribution['arcsine'](a) )
ureal(1,1.06066017177982,inf)
```

The names are (case-sensitive):

- 'gaussian'
- 'uniform'
- 'triangular'
- 'arcsine' or 'u_shaped'
- 'uniform_ring'
- 'uniform_disk'

8.3.4 Module contents

uniform(*a*)

Return the standard uncertainty for a uniform distribution.

Parameters **a** – the half-width

Example:

```
>>> x = ureal(1,type_b.uniform(1))
>>> summary(x)
'1.00, u=0.58, df=inf'
```

triangular(*a*)

Return the standard uncertainty for a triangular distribution.

Parameters **a** – the half-width

Example:

```
>>> x = ureal(1,type_b.triangular(1))
>>> summary(x)
'1.00, u=0.41, df=inf'
```

u_shaped(*a*)

Return the standard uncertainty for an arcsine distribution.

Parameters **a** – the half-width

Note: *arcsine* and *u_shaped* are equivalent

Example:

```
>>> x = ureal(1,type_b.arcsine(1))
>>> summary(x)
'1.00, u=0.71, df=inf'
```

arcsine(*a*)

Return the standard uncertainty for an arcsine distribution.

Parameters **a** – the half-width

Note: *arcsine* and *u_shaped* are equivalent

Example:

```
>>> x = ureal(1,type_b.arcsine(1))
>>> summary(x)
'1.00, u=0.71, df=inf'
```

uniform_ring(*a*)

Return the standard uncertainty for a uniform ring

Parameters **a** – the radius

Convert the radius `a` of a uniform ring distribution (in the complex plane) to a standard uncertainty

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

Example:

```
>>> z = ucomplex( 0, type_b.uniform_ring(1) )
>>> summary(z)
'(0.00+0.00j), u=[0.71,0.71], r=0.00, df=inf'
```

uniform_disk(*a*)

Return the standard uncertainty for a uniform disk

Parameters **a** – the radius

Convert the radius `a` of a uniform disk distribution (in the complex plane) to a standard uncertainty.

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

Example:

```
>>> z = ucomplex( 0, type_b.uniform_disk(1) )
>>> summary(z)
'(0.00+0.00j), u=[0.50,0.50], r=0.00, 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) )
>>> summary(X)
'(0.000+0.000j), u=[0.014,0.014], r=0.000, df=inf'
```

8.4 Reporting functions

This module provides functions that facilitate the reporting of information about `GTC` results.

The prefix *reporting* (or the alias `rp`) is needed to resolve objects defined in this module.

8.4.1 Reporting functions

- The function *budget* generates an uncertainty budget.
- The function *round* rounds values according to the level of uncertainty. It facilitates string formatting for uncertain numbers.
- The function *uncertainty_interval* calculates the upper and lower bounds of an uncertainty interval.
- The function *uncertainty_region* calculates a set of points located on the perimeter of an elliptical uncertainty region.
- The function *eigenv* evaluates the eigenvalues and eigenvectors of a 2-by-2 variance-covariance matrix.
- The function *k_factor* returns the coverage factor used for real-valued problems.
- 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.
- Functions *u_bar* and *v_bar* return summary values for matrices associated with two-dimensional uncertainty.
- The function *mahalanobis_sq* evaluates the squared Mahalanobis distance.

8.4.2 Coordinate changes

- Functions *u_polar_to_rect* and *u_rect_to_polar* transform statements of uncertainty from one system of coordinates to the other (polar and rectangular).
- Functions *u_rect_to_tangent* and *u_tangent_to_rect* transform statements of uncertainty between rectangular coordinates and tangential coordinates (in-phase / quadrature).
- The function *rotate_cv_coordinates* applies a rotation to the coordinate axes associated with a particular variance-covariance matrix.

8.4.3 Uncertainty functions

- The function *u_component* returns the component of uncertainty in one uncertain number due to uncertainty in another.
- The function *variance_and_dof* evaluates the variance and the degrees-of-freedom for both real and complex uncertain numbers.

8.4.4 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.

8.4.5 Module contents

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

Return a sequence of label-component of uncertainty pairs

Parameters

- **x** (`UncertainReal` or `UncertainComplex`) – the measurand estimate

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

A sequence of namedtuple pairs is returned, with the attributes `label` and `u`.

Each element is a pair: a label and the magnitude of the component of uncertainty (see *component*).

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

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

The argument `reverse` controls the sense of ordering.

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

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

Example:

```
>>> x1 = ureal(1,1,label='x1')
>>> x2 = ureal(2,0.5,label='x2')
>>> x3 = ureal(3,0.1,label='x3')
>>> y = (x1 - x2) / x3
>>> for l,u in reporting.budget(y):
...     print "%s: %G" % (l,u)
...
x1: 0.333333
x2: 0.166667
x3: 0.0111111

>>> for l,u in reporting.budget(y,reverse=False):
...     print "%s: %G" % (l,u)
...
x3: 0.0111111
x2: 0.166667
x1: 0.333333
```

u_component (*y*, *x*)

Return the component of uncertainty in `y` due to `x`.

Note:

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

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

•Otherwise, return 0.

Example:

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

>>> q = ucomplex(2,1)
```

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

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

round(*un*, *digits=2*, *df_decimals=0*)

Round the value of attributes of `un` ready for printing

If `un` is uncertain real, an `RoundedUncertainReal` object is returned with attributes `x`, `u`, `df`, `label` and `precision`.

The first four attributes correspond to the attributes of `un`. The values of `x` and `u` have been rounded to give *digits* significant figures in the uncertainty.

The degrees-of-freedom is rounded down to *df_decimals* decimal places.

`precision` is the number of decimal places needed to represent `x` and `u` in fixed-point format. It is intended for use when formatting a Python string.

Example

```
>>> un = ureal(10.2523,1.51,3.2,label='x')
>>> un_ = reporting.round(un)
>>> "{0.label!s}: {0.x:.{0.precision}f},        ... u={0.u:.{0.precision}f},␣
↪df={0.df:.0f}".format( un_ )
x: 10.3, u=1.5, df=3
```

sensitivity(*y*, *x*)

Return the partial derivative of `y` with respect to `x`.

Note:

- If `x` and `y` are uncertain real numbers, return a float.
- **If one of `y` or `x` is an uncertain complex number, return** a 4-element sequence of float, containing the elements of the Jacobian matrix.
- Otherwise, return 0.

Example:

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

>>> q = ucomplex(2,1)
>>> r = ucomplex(3,1)
>>> z = q * r
>>> reporting.sensitivity(z,q)
jacobian_matrix(rr=3.0, ri=0.0, ir=0.0, ii=3.0)

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

k_factor (*df=inf*, *p=95*, *quick=True*)
Return the univariate coverage factor

Parameters

- **df** (*float*) – the degrees-of-freedom (>1)
- **p** (*integer*) – the coverage probability (%)
- **quick** (*bool*) – use faster algorithm

Evaluates the coverage factor for an uncertainty interval with coverage probability `p` and degrees-of-freedom `df` based on Student's t-distribution.

Accuracy is reduced when using the *quick* option, for example:

```
>>> reporting.k_factor(3)
3.181632923406346
>>> reporting.k_factor(3,quick=False)
3.1824463052837175
```

k_to_dof (*k*, *p=95*)
Return the dof for *k* a univariate coverage factor

Parameters

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

Evaluates the degrees-of-freedom given a coverage factor for an uncertainty interval with coverage probability `p` based on Student's t-distribution.

Returns infinity for degrees-of-freedom values above 1E8.

Example:

```
>>> reporting.k_to_dof(2.0,95)
60.437564504892904
```

uncertainty_interval (*x*, *p=95*, *quick=True*)
Return the upper and lower bounds of a *p%* uncertainty interval.

Parameters

- **x** (`UncertainReal`) – the value (estimate)
- **p** (*integer*) – coverage probability in percent
- **quick** (*bool*) – use the faster coverage factor algorithm

Returns a 2-element sequence

Example:

```
>>> x = ureal(1.5,1,50)
>>> reporting.uncertainty_interval(x)
expanded_uncertainty(lower=-0.5085590722581137, upper=3.5085590722581137)
```

is_ureal (*x*)
Return `True` if `x` is an uncertain real number

Example:

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

is_ucomplex(*z*)

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

Example:

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

variance_and_dof(*x*)

Return the variance and degrees-of-freedom.

If `x` is an uncertain real number, a pair of real numbers is returned `(v,df)`, where `v` is the standard variance and `df` is the degrees-of-freedom calculated using the Welch-Satterthwaite formula.

If `x` is an uncertain complex number, a sequence and a float is returned `(cv,df)`, where `cv` is a 4-element sequence representing the variance-covariance matrix and `df` is the degrees-of-freedom, calculated using the Willink-Hall total-variance method.

Otherwise, returns `(0.0,inf)`.

Example:

```
>>> x1 = ureal(1.1,1,5)
>>> x2 = ureal(2.3,1,15)
>>> x3 = ureal(-3.5,1,50)
>>> y = (x1 + x2) / x3
>>> v,df = reporting.variance_and_dof(y)
>>> v
0.24029987505206163
>>> df
30.460148613530492
```

u_to_cv(*u_re_im*, *r=0*)

Convert standard uncertainties to a covariance matrix

Parameters

- **u_re_im** (*a 2-element sequence of float, or a float*) – standard uncertainties for the real and imaginary components
- **r** (*float*) – correlation coefficient between the real and imaginary components

Returns (v_rr,v_ri,v_ir,v_ii) : the four elements of the covariance matrix

Example:

```
>>> reporting.u_to_cv( (0.1,0.1) ,0.5)
variance_covariance(rr=0.010000000000000002, ri=0.005000000000000001,
ir=0.005000000000000001, ii=0.010000000000000002)

>>> reporting.u_to_cv(3)
variance_covariance(rr=9, ri=0, ir=0, ii=9)
```

cv_to_u(*cv*)

Return standard uncertainties and a correlation coefficient

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

Returns (u_re,u_im), r : the standard uncertainties and a correlation coefficient

A pair of standard uncertainties in the real and imaginary components and a correlation coefficient are obtained from the variance-covariance matrix `cv`.

Example:

```
>>> reporting.cv_to_u( (0.01,0.005,0.005,0.01) )
(standard_uncertainty(real=0.1, imag=0.1), 0.4999999999999999)
```

v_bar(*cv*)

Return the trace of `cv` divided by 2

Parameters **cv** (*a 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
variance_covariance(rr=2.3464, ri=1.8432, ir=1.8432, ii=51.4216)
>>> reporting.v_bar(y.v)
26.884
```

u_bar(*ucpt*)

Return the magnitude of a component of uncertainty.

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

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

If `ucpt` is a number, return the magnitude.

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
u_components(rr=0.2, ri=-7.1, ir=7.1, ii=0.2)
>>> reporting.u_bar(dy_dz1)
7.102816342831905
```

fn_bar(*ucpt*)

Deprecated function name: use *u_bar* instead

tv_bar(*cv*)

Deprecated function name: use *v_bar* instead

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

Return the bivariate coverage factor squared

Parameters

- **df** (*float*) – the degrees-of-freedom (>=2)
- **quick** (*bool*) – use faster algorithm

Arg p: 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.

The `quick` option is not available for *df* less than 3

Accuracy is reduced when using the *quick* option, for example:

```
>>> reporting.k2_factor_sq(3)
57.35717041567613
>>> reporting.k2_factor_sq(3,quick=False)
56.99999999936168
```

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

Return the dof for *k2* a bivariate coverage factor

Parameters

- **k2** – coverage factor (>0)
- **p** – 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.

Returns infinity for degrees-of-freedom values above 1E8.

Example:

```
>>> reporting.k2_to_dof(2.6,95)
34.357884313812384
```

mahalanobis_sq(*x*, *y*, *cv*, *inverted=False*)

Return the squared Mahalanobis distance between `x` and `y`

Parameters

- **x** – complex number
- **y** – complex number
- **cv** (*4-element sequence of float*) – variance-covariance matrix
- **inverted** (*Boolean*) – True if *cv* is inverted

If `inverted` is `True`, the matrix inverse of `cv` is not calculated.

Example:

```
>>> cv = reporting.u_to_cv( (2,4) )
>>> x1 = complex(1,3)
>>> x2 = complex(3,1)
>>> y = complex(1,1)
>>> reporting.mahalanobis_sq(x1,y,cv)
0.25
>>> reporting.mahalanobis_sq(x2,y,cv)
1.0
```

uncertainty_region(*N*, *z*, *p=95*, *quick=True*)

Return a sequence of N (x,y) points on a *p*% uncertainty ellipse

Parameters

- **N** (*int or long*) – number of coordinates required
- **z** (`UncertainComplex`) – the value (estimate)
- **p** (*integer*) – coverage probability in percent
- **quick** (*bool*) – use the faster coverage factor algorithm

This function generates `N` points on the perimeter of an elliptical uncertainty region.

Example:

```
>>> z = ucomplex(0,1,50)
>>> reporting.uncertainty_region(10,z)
[(2.53924593344018+0j),
(1.9451752370243123+1.6321958239622785j),
(0.44093542899005095+2.50066908205661j),
(-1.2696229667200893+2.199051484815526j),
(-2.3861106660143627+0.8684732580943322j),
(-2.386110666014363-0.8684732580943316j),
(-1.2696229667200911-2.199051484815525j),
(0.44093542899004984-2.5006690820566106j),
(1.9451752370243116-1.6321958239622794j),
(2.53924593344018-6.219358809270899e-16j)]
```

eigenv(*cv*)

Return the eigenvalues and eigenvectors of `cv`

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

Returns a pair of eigenvalues and a pair of eigenvectors

The function raises a `RuntimeError` exception if the off-diagonal elements differ by more than 1E-10, or if either diagonal element is negative.

Calculates the eigenvalues and eigenvectors of a variance-covariance matrix `cv`.

Returns a pair of eigenvalues (ordered) and a pair of eigenvectors (2-element sequences). The eigenvectors have unit magnitude.

Examples:

```
>>> z = ucomplex(1+1j,(.2,.4))
>>> L,V = reporting.eigenv(z.v)
>>> L
(0.16000000000000003, 0.04000000000000001)
>>> V
((0, 1), (1, 0))

>>> m = linear_algebra.matrix( [[.8,.3],[.3,.7]])
>>> L,V = reporting.eigenv(m.flat)
>>> L
(1.054138126514911, 0.445861873485089)
>>> V[0]
(0.7630199824727258, 0.6463748961301957)
>>> V[1]
(-0.6463748961301957, 0.7630199824727257)
```

u_polar_to_rect(*z*, *u_r_phi*, *r=0*)

Return standard uncertainties, and correlation, in rectangular coordinates

Parameters

- **z** (*complex*) – the value (estimate)
- **u_r_phi** (*2-element sequence of float*) – a pair of standard uncertainties in polar coordinates (angle uncertainty in radians)
- **r** (*float*) – correlation between the radial and azimuthal components

Returns (u_re,u_im), r : a pair of standard uncertainties and a correlation coefficient in rectangular coordinates

Transform a pair of standard uncertainties in polar coordinates into a pair of standard uncertainties in rectangular coordinates.

Note: If `u/abs(z)< 5` the polar to rectangular conversion is not recommended.

Example:

```
>>> z = complex(0.93,-0.03)
>>> u_r_phi = (0.01,math.radians(.5))
>>> u,r = reporting.u_polar_to_rect(z,u_r_phi)
>>> u
standard_uncertainty(real=0.009998229284003339, imag=0.008122182692929972)
>>> r
-0.013517808883928624
```

u_rect_to_polar(*z*, *u_re_im*, *r=0*)

Return standard uncertainties, and correlation, in polar coordinates

Parameters

- **z** (*complex*) – the value (estimate)
- **u_re_im** (*2-element sequence of float*) – a pair of standard uncertainties in rectangular coordinates
- **r** (*float*) – correlation between the real and imaginary components

Returns (u_r,u_phi), r : a pair of standard uncertainties and a correlation coefficient in polar coordinates (angle uncertainty in radians)

Transform a pair of standard uncertainties in rectangular coordinates into a pair of standard uncertainties in polar coordinates.

Note: If `u/abs(z)< 5` the rectangular to polar conversion is not recommended.

Example:

```
>>> z = complex(0.93,-0.03)
>>> u_re_im = (0.05,0.05)
>>> u_p,r = reporting.u_rect_to_polar(z,u_re_im)
>>> u_p
(0.05000000000000001, 0.05373549001826343)
>>> r
0
```

u_rect_to_tangent(*z*, *u_re_im*, *r=0*)

Return standard uncertainties and correlation in tangential coordinates

Parameters

- **z** (*complex*) – the value (estimate)
- **u_re_im** (*2-element sequence of float*) – a pair of standard uncertainties for the real and imaginary components of `z`
- **r** (*float*) – correlation between components

Returns (u_r,u_t), r_t : a pair of uncertainties expressed in coordinates aligned with the polar axes and a correlation coefficient

Transform a pair of standard uncertainties in rectangular coordinates into a pair of uncertainties expressed in rotated rectangular coordinates (tangential coordinates) aligned with the polar axes at the point `z`.

Example:

```
>>> z = complex(1,1)
>>> u = (1,2)
>>> u_rt, r = reporting.u_rect_to_tangent(z, u )
>>> u_rt
rt_uncertainty(radial=1.5811388300841895, tangent=1.5811388300841898)
>>> u_rt.radial
1.5811388300841895
>>> u_rt.tangent
1.5811388300841898
>>> r
0.6
```

Note: Raises a RuntimeError if `abs(z) == 0`

u_tangent_to_rect(*z*, *u_rt*, *r=0*)

Return standard uncertainties and the correlation coefficient

Parameters

- **z** (*complex*) – the value (estimate)
- **u_rt** (*2-element sequence of float*) – a pair of standard uncertainties given in tangential coordinates
- **r** (*float*) – correlation between the radial and tangential components

Returns (u_re,u_im), r : a pair of standard uncertainties and a correlation coefficient in rectangular coordinates

Transform a pair of standard uncertainties and a correlation coefficient in tangential coordinates (aligned to the polar axes at the point `z`) to a pair of standard uncertainties and a correlation coefficient expressed in rectangular coordinates.

Example:

```
>>> z = complex(.3,.4)
>>> u_rt = (.5,1)
>>> u,r = reporting.u_tangent_to_rect(z, u_rt)
>>> u
standard_uncertainty(real=0.8544003745317532, imag=0.7211102550927979)
>>> u.real
0.8544003745317532
>>> u.imag
0.7211102550927979
>>> r
-0.5843047258450758
```

rotate_cv_coordinates(*cv*, *phi*)

Return the covariance matrix in a rotated coordinate system.

Parameters

- **cv** (*4-element sequence of float*) – initial covariance matrix
- **phi** (*float*) – angle of rotation (radians)

Returns covariance matrix in rotated coordinate system

The coordinate axes associated with the covariance matrix `cv` are rotated counter-clockwise through an angle `phi`.

Example:

```
>>> z = ucomplex(1,(2,1))
>>> reporting.rotate_cv_coordinates(z.v,math.pi/4)
variance_covariance(rr=2.5000000000000004, ri=1.5, ir=1.5, ii=2.
↪4999999999999996)
```

8.5 Linear algebra

This module supports array and matrix operations.

The prefix *linear_algebra* (or the alias `la`) is needed as to resolve the names of objects defined in this module.

8.5.1 Classes

- *array*
- *matrix*

8.5.2 Arithmetic operations

Arithmetic operations are defined for both arrays and matrices (unary + and −, and binary +, − and ⋆). The multiplication operator ⋆ is implemented element-wise for arrays, but performs the matrix product for matrices.

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

array and *matrix* define attributes as short-hand for some operations:

- the *array*, or *matrix*, attribute `T` transposes the object
- the *matrix* attribute `I` evaluates the matrix inverse

8.5.3 Functions

The functions *inverse*, *transpose*, *solve* and *det* implement the usual linear algebra operations.

Functions *asmatrix* and *asarray* change the type of object. The function *aslist* turns an array or matrix into a (possibly nested) list.

The functions *identity*, *empty*, *zeros* and *ones* create simple arrays.

8.5.4 Array broadcasting

When binary arithmetic operations are used on 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 larger dimension size is used. For example:

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

8.5.5 Module contents

class **array**(*obj*, *copy=True*)

An array object stores a regular array of elements

Parameters

- **obj** – an array, a matrix, a sequence or an iterable object
- **copy** – Boolean

An *array* can be constructed from another *array* or *matrix*, a sequence or an iterator. Nested sequences or iterators create multidimensional arrays.

When `copy` is `True` and `obj` is an *array* or, *matrix*, data is shared, not copied.

Examples:

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

>>> x = la.array([[0, 1],[2, 3]])
>>> x
array([[0, 1],
[2, 3]])
```

T

The transpose

(see also *transpose*)

Example:

```
>>> x = la.array( range(8))
>>> x.shape = 2,4
>>> xt = x.T
>>> xt.shape
(4, 2)
>>> print x
[[0 1 2 3]
[4 5 6 7]]
>>> print xt
[[0 4]
[1 5]
[2 6]
[3 7]]
```

copy()

Return a copy

Example:

```
>>> a = array( [1,2] )
>>> a_copy = a.copy()
>>> a_copy
array([1, 2])
>>> a
array([1, 2])
>>> a[1] = 3
>>> a_copy
array([1, 2])
>>> a
array([1, 3])
```

flat
An iterator for all elements

Example:

```
>>> x = la.array( [[1,2],[3,4]])
>>> print x
[[1 2]
[3 4]]
>>> xf = la.array( x.flat )
>>> print xf
[1 2 3 4]
```

shape
a tuple of dimension sizes

class matrix (*obj*, *copy=True*)
A matrix stores a 2-by-2 array of objects

Parameters

- **obj** – an array, a matrix, a sequence or an iterable
- **copy** – Boolean

A *matrix* may be constructed from an *array* or *matrix*, a sequence or an iterator.

When `copy` is `True` and `obj` is an *array*, or *matrix*, data is shared, not copied.

Examples:

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

>>> x = la.matrix([[0, 1],[2, 3]])
>>> x
matrix([[0, 1],
[2, 3]])
```

I
The matrix inverse

(see also *inverse*)

Example:

```
>>> x = la.matrix( [[1,2],[3,4]])
>>> x * x.I
```

```
matrix([[0.9999999999999998, 1.1102230246251565e-16],
[0.0, 1.0000000000000002]])
```

T

The transpose

(see also *transpose*)

Example:

```
>>> x = la.array( range(8))
>>> x.shape = 2,4
>>> xt = x.T
>>> xt.shape
(4, 2)
>>> print x
[[0 1 2 3]
[4 5 6 7]]
>>> print xt
[[0 4]
[1 5]
[2 6]
[3 7]]
```

copy()

Return a copy

Example:

```
>>> a = array( [1,2] )
>>> a_copy = a.copy()
>>> a_copy
array([1, 2])
>>> a
array([1, 2])
>>> a[1] = 3
>>> a_copy
array([1, 2])
>>> a
array([1, 3])
```

flat

An iterator for all elements

Example:

```
>>> x = la.array( [[1,2],[3,4]])
>>> print x
[[1 2]
[3 4]]
>>> xf = la.array( x.flat )
>>> print xf
[1 2 3 4]
```

shape

a tuple of dimension sizes

det(*a*)

Return the matrix determinant

Example:

```
>>> x = la.matrix( range(4) )
>>> x.shape = 2,2
```

```
>>> print x
[[0 1]
[2 3]]
>>> la.det(x)
-2.0
```

inverse(*a*)
Return the matrix inverse

Example:

```
>>> x = la.matrix( [[2,1],[3,4]])
>>> x * la.inverse(x)
matrix([[1.0, 0.0],
[4.440892098500626e-16, 1.0]])
```

transpose(*a*, *axes=None*)
Return the transpose of `a`.

Parameters

- **a** – an array, or matrix
- **axes** – a sequence of axis indices

Returns an array, or matrix (the same type as `a`).

Note:

•A new object is created (ie, `a` is not affected).

•By default, the function will reverse the dimensions of `a`.

•`axes` may contain a sequence of integers indexing the dimensions of `a`. The axes of `a` will then be permuted according to the contents of `axes`.

Examples:

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

>>> x = la.array( range(12) )
>>> x.shape = 2,3,2
>>> print x
[[[0 1]
[2 3]
[4 5]]
[[6 7]
[8 9]
[10 11]]]
>>> la.transpose(x,(0,2,1))
array([[[0, 2, 4],
[1, 3, 5]],
[[6, 8, 10],
[7, 9, 11]]])
```

identity(*ndim*)
Return an identity array with `ndim` dimensions

Example:

```
>>> la.identity(3)
array([[1.0, 0, 0],
[0, 1.0, 0],
[0, 0, 1.0]])
```

ones (*shape*)
Return an array of shape `shape` containing `1`'s

Example:

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

empty (*shape*)
Return an array of shape `shape` containing `None` elements

Example:

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

zeros (*shape*)
Return an array of shape `shape` containing `0`'s

Example:

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

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

Parameters

- **a** – an array or matrix
- **b** – an array or matrix

Return type array or matrix

The type of object returned depends on the type of arguments. A matrix is returned if `a` or `b` is a matrix, otherwise an array is returned.

Example:

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

asarray (a)
Return `a` as an array without copying

Example:

```
>>> x = la.array( (1,2,3) )
>>> x
array([1, 2, 3])
>>> la.asmatrix(x)
matrix([[1, 2, 3]])
```

asmatrix(*a*)
Return a as a matrix without copying

Example:

```
>>> x = la.array( range(4) )
>>> la.asmatrix(x)
matrix([[0, 1, 2, 3]])
```

aslist(*a*)
Return a list of the data in a

A list is constructed by iterating over the elements of a. If the elements of a are iterable, a nested list is created.

Parameters **a** – an iterable object

8.6 Conversion between numbers and strings

The prefix *number_strings* (or the alias ns) is needed as to resolve the names of objects defined in this module.

8.6.1 Loss of precision

Problems can arise when converting between numbers and strings. Python conversions can truncate the number of floating-point digits, which introduces errors.

The functions in this module can be used to avoid unintended numerical errors.

8.6.2 Functions

- to_string(x) : converts x to a string
- sequence_printer(seq) : uses to_string(x) to convert the elements of a sequence
- to_numeric(x) : converts x to a number
- sequence_parser(seq) : uses to_numeric(x) to convert the elements of a sequence

8.6.3 Module contents

class File (*filename*, *mode*, *delim=None*)
Provides methods for reading and writing numeric data

Numbers are converted to, or from, text without loss of precision.

Other functionality defaults to standard Python file behaviour.

Examples:

```
# read a file
with ns.File('my_file','r',delim=',') as file:
    for line in file: print line

# or ...
file = ns.File('my_file','r',delim=',')
for line in file: print line
```

newline()
Put a newline character in the file

read(*delim=None*)
Return file contents as a sequence

All strings are converted to numbers, if possible

`delim` separates elements in the line (`delim` is set when opening the file [default: whitespace])

readline(*delim=None*)
Return one line as a sequence

All strings are converted to numbers, if possible

`delim` separates elements in the line [default: whitespace] (`delim` can also be set when opening the file)

readlines(*delim=None*)
Return file as a sequence of line sequences

All strings are converted to numbers, if possible

`delim` separates elements in the line [default: whitespace] (`delim` can also be set when opening the file)

write(*seq*, *delim=None*)
Write `seq` to the file

Numbers in `seq` are converted to strings without loss of precision.

`write` does not add a line separator at the end of the write sequence (use *newline*).

to_string(*x*)
Return a string representing `x`

Parameters **x** – anything

Returns a string representation of `x`

The string format generated for complex numbers is recognised by spreadsheet applications.

For non-numeric types `repr` is used to generate a string.

Examples:

```
>>> z = 1.0/3.0 - 0.3134j
>>> ns.to_string(z)
'0.3333333333333333-0.3134j'

>>> x = float(3.141414E-21)
>>> ns.to_string(x)
'3.141414e-21'
```

sequence_printer(*seq*)
Return a list of strings

Parameters **seq** – a sequence, or nested sequences

Returns a list of strings corresponding to of the elements in `seq`

Numeric elements in `seq` retain the internal floating point precision (see *to_string*).

If `seq` contains nested sequences, the elements will be converted recursively and the lists returned will also be nested.

Example:

```
>>> x = la.array( [ 0.123 * i for i in range(12) ])
>>> x.shape = 2,2,3
>>> print x                  # Python string conversion limits precision
[[[0.0 0.123 0.246]
[0.369 0.492 0.615]]
```

```
[[0.738 0.861 0.984]
[1.107 1.23 1.353]]]
>>> ns.sequence_printer(x)      # precision conserved
[[['0.0', '0.123', '0.246'],
['0.369', '0.492', '0.615']],
[['0.738', '0.861', '0.984'],
['1.107', '1.23', '1.353']]
]
```

to_numeric(*x*)

Return x as a number, if possible

Parameters **x** – string

Returns a number, if possible, otherwise x

Examples:

```
>>> x = '3.141413999999999e-21'
>>> ns.to_numeric(x)
3.141413999999999e-21

>>> z = '0.3333333333333333-0.3134j'
>>> ns.to_numeric(z)
(0.3333333333333333-0.3134j)
```

sequence_parser(*seq*)

Return a list of numbers

Parameters **seq** – a sequence of strings

Returns a list of strings and numbers

Elements in seq are converted to numbers where possible, otherwise the original is returned (see *to_numeric*).

If seq contains nested sequences, elements will be converted recursively and nested sequences are returned.

Example:

```
>>> x = [['0.0', '0.123', '0.246'],
... ['0.3689999999999999', '0.4919999999999999', '0.6149999999999999']]
>>> ns.sequence_parser(x)
[[0.0, 0.123, 0.246], [0.3689999999999999, 0.4919999999999999, 0.
↪6149999999999999]]
```

open(*filename*, *mode=None*, *delim=None*)

Return a *File* object for handling numeric data

Parameters

- **filename** – file name
- **mode** – opening mode
- **delim** – file delimiting character

Note: This function is deprecated. Use File instead, or use use File in a with ... as statement.

Example:

```
# open CVS file and print contents
>>> file = ns.open('my_file','r',delim=',')
>>> for line in file:
```

```
...     print line
...
```

8.7 Additional functions

The prefix *function* (or the alias fn) is needed as to resolve the names of objects defined in this module.

8.7.1 Coordinate transformation

Functions *polar* and *rect* provide polar and rectangular coordinate transformations for uncertain numbers.

8.7.2 Implicit problems

The function *implicit* solves problems of the form:

$fn(x) = 0$

obtaining the solution x as an uncertain real number.

8.7.3 Utility functions

The functions *complex_to_seq* and *seq_to_complex* support a matrix representation of complex numbers.

The function *mul2* can be used when a product of a pair of uncertain numbers is close to zero. The evaluation of uncertainty components includes a second-order contribution to the uncertainty.

8.7.4 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.

8.7.5 Module contents

polar(*z*)

Return a pair of uncertain real numbers for magnitude and phase.

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

Returns uncertain real numbers for magnitude and phase, a 2-element sequence of UncertainReal

Example:

```
>>> z = ucomplex(0.95+0.02j,0.01)
>>> z
ucomplex((0.9499999999999996+0.02j), u=[0.01,0.01], r=0, df=inf)
>>> m,p = function.polar(z)
>>> m
```

```
ureal(0.95021050299394183,0.01,inf)
>>> p
ureal(0.021049522137046431,0.010523983863040671,inf)

>>> z_mp = function.polar(z)
>>> z_mp.magnitude
ureal(0.95021050299394183,0.01,inf)
>>> z_mp.phase
ureal(0.021049522137046431,0.010523983863040671,inf)
```

rect(*mag_phase*)

Return an uncertain complex number in rectangular coordinates.

Parameters **mag_phase** (2-element sequence of `UncertainReal`) – a sequence containing the magnitude and phase (radians)

Returns `UncertainComplex`

Example:

```
>>> phi = ureal( math.radians(87), math.radians(.5) )
>>> mag = ureal( 0.87, 0.07)
>>> z = function.rect( (mag,phi) )
>>> z.s
'(0.0455+0.8688j), u=[0.0084,0.0699], r=0.4300, df=inf'
```

complex_to_seq(*z*)

Transform a complex number into a 4-element sequence

Parameters **z** – a number

If `z = x + yj`, then `matrix([[x,-y],[y,x]])` can be used to represent `z` in matrix computations.

Examples:

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

>>> m = linear_algebra.matrix( function.complex_to_seq(z) )
>>> m.shape = (2,2)
>>> print( m )
[[1.0 -2.0]
[2.0 1.0]]
```

seq_to_complex(*seq*)

Transform a 4-element sequence, or array, into a complex number

Parameters **seq** – a 4-element sequence, *array* or *matrix*

A `RuntimeError` will be raised if `seq` is ill-conditioned.

If `z = x + yj`, then `matrix([[x,-y],[y,x]])` can be used to represent `z` in matrix computations.

Examples:

```
>>> seq = (1,-2,2,1)
>>> z = function.seq_to_complex( seq )
>>> print( z )
(1+2j)

>>> a = linear_algebra.array((1,-2,2,1))
>>> a.shape = 2,2
>>> print(a)
[[1 -2]
[2 1]]
```

```
>>> z = function.seq_to_complex(a)
>>> print( z )
(1+2j)
```

mean (*seq*)
Return the arithmetic mean of elements in *seq*

seq - an iterable object

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

Example

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

line_fit (*x*, *y*)
-> Least-squares linear intercept and slope

Parameters

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

Returns a *LineFitOLS* object

y must be a sequence of uncertain real numbers.

Performs an ordinary least-squares regression.

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

The function *type_a.line_fit* 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_components*).

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 = fn.line_fit(x,y).a_b
>>> a
ureal(10,0.1175507627290518,inf)
>>> b
ureal(-3,0.022019275302527213,inf)
```

line_fit_wls (*x*, *y*, *u_y=None*)
-> Weighted least-squares linear regression

Parameters

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

Returns a *LineFitWLS* object

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_components*).

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 = function.line_fit_wls(x,y)
>>> a, b = fit.a_b
>>> a
ureal(0.8852320675105498,0.5297081435088364,inf)
>>> b
ureal(2.056962025316455,0.177892016741205,inf)
```

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

Perform straight-line regression with uncertainty in x and y

Parameters

- **a_b** – a pair of initial estimates for a and b
- **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
- **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. The uncertainties of the x and y sequences are used to calculate weights for the regression unless the optional arguments u_x and u_y are not specified.

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 itertools.izip(xin,uxin) ]
y = [ ureal(yin_i,uyin_i) for yin_i,uyin_i in itertools.izip(yin,uyin) ]

# initial estimate
a_b = function.line_fit(x,y).a_b

# TLS returns uncertain numbers
a,b = function.line_fit_wtls(a_b,x,y).a_b
```

class **LineFitOLS** (*a*, *b*, *ssr*, *N*)
This object holds results from an ordinary linear regression to data.

N
The number of points in the sample

a_b
Return the intercept and slope as uncertain numbers

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.

class **LineFitWLS** (*a*, *b*, *ssr*, *N*)
This object holds results from a weighted LS linear regression to data.

N
The number of points in the sample

a_b
Return the intercept and slope as uncertain numbers

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.

class **LineFitWTLS** (*a*, *b*, *ssr*, *N*)
This object holds results from a TLS linear regression to data.

N
The number of points in the sample

a_b
Return the intercept and slope as uncertain numbers

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.

mul2 (*arg1*, *arg2*, *estimated=False*)
Return the product of `arg1` and `arg2`

Extends the usual calculation of a product, by using second-order contributions to uncertainty.

Parameters

- **arg1** – uncertain real or complex number
- **arg2** – uncertain real or complex number
- **estimated** – Boolean

When both arguments are uncertain numbers that always have the same fixed values then `estimated` should be set `False`. For instance, residual errors are often associated with the value 0, or 1, which is not measured, in that case `estimated=False` is appropriate. However, if either or both arguments are based on measured values set `estimated=True`.

Note: When `estimated` is `True`, and the product is close to zero, the result of a second-order uncertainty calculation is smaller than the uncertainty calculated by the usual first-order method. In some cases, an uncertainty of zero will be obtained.

There are fairly strict limitations on the use of this function, especially for uncertain complex numbers:

1) Arguments must be independent (have no common influence quantities) and there can be no correlation between any of the quantities that influence *arg1* or *arg2*.

2) If either argument is uncertain complex, the real and imaginary components must have equal uncertainties (i.e., the covariance matrix must be diagonal with equal elements along the diagonal) and be independent (no common influences).

A `RuntimeError` exception is raised if these conditions are not met.

Note: This function has been developed to improve the accuracy of uncertainty calculations where one or both multiplicands are zero. In such cases, the usual method of uncertainty propagation fails.

For example

```
>>> x1 = ureal(0,1,label='x1')
>>> x2 = ureal(0,1,label='x2')
>>> y = x1 * x2
>>> y
ureal(0,0,inf)
>>> for cpt in rp.budget(y,trim=0):
...         print "  %s: %G" % cpt
...
  x1: 0
  x2: 0
```

so none of the uncertainty in `x1` or `x2` is propagated to `y`. However, we may calculate the second-order contribution

```
>>> y = fn.mul2(x1,x2)
>>> y
ureal(0,1,inf)
>>> for cpt in rp.budget(y,trim=0):
...         print "  %s: %G" % cpt
...
  x1: 0.707107
  x2: 0.707107
```

The product now has a standard uncertainty of unity.

Warning: *mul2* departs from the first-order linear calculation of uncertainty in the GUM.

In particular, the strict proportionality between components of uncertainty and first-order partial derivatives no longer holds.

implicit (*fn*, *x_min*, *x_max*, *epsilon=2.220446049250313e-16*)
Return the solution to $f(x) = 0$

Parameters

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

The user-defined function `fn` takes a single uncertain real number argument.

`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(1,0.025000000000000001,inf)
```

8.8 Storing uncertain numbers

Archiving allows uncertain numbers to be saved and restored (for further calculation).

The archiving process preserves the identity of uncertain numbers. For example, if a particular reference standard has been used in a number of different measurements, the dependence on common influences is recorded.

Note: Archives created with GTC version 0.9.7 are not compatible with this version.

The prefix *archive* (or the alias `ar`) must be used to resolve the names of objects defined in this module.

8.8.1 Class

An *Archive* object marshals a set of uncertain numbers for storage and recreates uncertain numbers, when restoring an archive.

8.8.2 Functions

An archive can be stored as a computer file, or in a string.

Functions for storing and retrieving an archive file are

- *load*

- *dump*

Functions for storing and retrieving string archives are

- *dumps*
- *loads*

8.8.3 Module contents

class Archive
Defines objects used to store and recover uncertain numbers

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

Example:

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

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

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

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

Example:

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

items()
Return a list of pairs of name-tags and uncertain-numbers

iteritems()
Return an iterator for pairs of name-tags and uncertain-numbers

iterkeys()
Return an iterator for name-tags

itervalues()
Return an iterator for uncertain numbers

keys()
Return a list of name-tags

values()
Return a list of uncertain numbers

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

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

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

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

Parameters

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

- **ar** – an *Archive* object

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

Note: This function can only be called once on a particular archive.

dumps(*ar*, *protocol=-1*)

Return a string representation of the archive

Parameters

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

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

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

Note: This function can only be called once on a particular archive.

loads(*s*)

Return an archive object restored from a string representation

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

8.9 Tools for validating uncertainty calculations

Functions in this module can be used to validate uncertainty calculations by a simulation method.

If an uncertainty calculation preforms well on simulated data, it can be expected to perform well with real data too.

What does 'perform well' mean? *Coverage probability* (level of confidence) can be understood as the success-rate when a procedure is applied many times to independent data sets. That is, calculated uncertainty statements, for different sets of experimental data, should cover the measurand on roughly 95 out of 100 occasions when the nominal level of confidence is 95%. A simulation should represent the main features of the data generated in actual measurements.

The functions in this module fall into two groups: those that simulate random effects and those that test whether the measurand is covered by the uncertainty statement. Both real and complex-valued measurands can be handled.

There are two groups of function that simulate random effects. One group generates random errors drawn from a particular error distribution. The other group generates a triplet of numbers needed to define uncertain numbers: the estimate, the uncertainty of the estimate and the degrees of freedom associated with the uncertainty.

A suffix in the function names distinguishes between the two types: `_err` indicates a simple random error generator (e.g., *gaussian_err*), `_est` indicates a generator of estimates (e.g., *gaussian_est*)

Generators of estimates are used to create independent inputs for an uncertainty calculation that varies from one measurement to the next.

The prefix *sim* is needed to resolve the names of objects defined in this module.

8.9.1 Functions that create simple random error generators (real-valued)

- *gaussian_err* - a Gaussian random error generator
- *uniform_err* - a uniform random error generator
- *triangular_err* - a triangular random error generator

- *arcsine_err* - an arcsine random error generator

In the case of correlated errors, the function *mv_gaussian_err*, which is associated with a multivariate Gaussian distribution, can be used.

8.9.2 Functions that generate estimates (real-valued)

- *gaussian_est* - an input associated with a Gaussian distribution
- *uniform_est* - an input associated with a uniform distribution
- *triangular_est* - an input associated with a triangular distribution
- *arcsine_est* - an input associated with an arcsine distribution

The generators created by these functions produce data triplets (x, u, df), which are convenient for direct input into GTC calculations.

In the case of correlated inputs the function *mv_gaussian_est*, which is associated with a multivariate Gaussian distribution, can be used.

8.9.3 Functions that create random error generators (complex-valued)

- *bi_gaussian_err* - a bivariate Gaussian error generator
- *uniform_ring_err* - a ring distribution generator
- *uniform_disk_err* - a disk distribution generator

8.9.4 Functions that generate estimates (complex-valued)

- *bi_gaussian_est* - an input associated with a bivariate Gaussian distribution

8.9.5 Utility functions

The mapping `error` selects random error generator functions by name (without the `_err` suffix).

The mapping `estimate` selects estimate generators functions by name (without the `_est` suffix).

The function `seed` gives access to the seed function of the Python *random* module.

The functions:

- *interval_OK*
- *circle_OK*
- *ellipse_OK*

return `True` when the measurand is contained by the uncertainty statement

Summing the number of successes provides an estimate of the coverage probability for a procedure. The standard deviation of this estimate is calculated by *success_rate_sd*.

8.9.6 Module contents

gaussian_err(*x*, *u*)

Return a generator of Gaussian errors

Parameters

- **x** – the mean

- **u** – the standard deviation

Example:

```
>>> gen = sim.gaussian_err(1,.5)
>>> gen.next()
1.2333329637100736
>>> gen.next()
-0.26827031760393716
```

uniform_err(*x*, *a*)

Return a generator of uniformly distributed errors

Parameters

- **x** – the mean
- **a** – half-width

The standard deviation of the simulated distribution is `u = a/sqrt(3)`.

Example:

```
>>> gen = sim.uniform_err(1,.5)
>>> gen.next()
0.7138525970522182
>>> gen.next()
1.115028001086297
```

triangular_err(*x*, *a*)

Return a generator of triangular distribution errors

Parameters

- **x** – the mean
- **a** – half-width

The standard deviation of the simulated distribution is `u = a/sqrt(6)`.

Example:

```
>>> gen = sim.triangular_err(1,.5)
>>> gen.next()
0.8490224542656288
>>> gen.next()
1.0586618711505276
```

arcsine_err(*x*, *a*)

Return a generator of arcsine distributed errors

Parameters

- **x** – the mean
- **a** – half-width

The standard deviation of the simulated distribution is `u = a/sqrt(2)`.

Example:

```
>>> gen = sim.arcsine_err(1,.5)
>>> gen.next()
1.285842415423328
>>> gen.next()
0.5577409002191889
```

gaussian_est(*x*, *u*, *df=inf*)

Return a generator of Gaussian input estimates

Parameters

- **x** – best estimate, used as the mean in simulations
- **u** – standard uncertainty, used as standard deviation in simulations
- **df** – degrees-of-freedom

The generator produces namedtuples with attributes `x`, `u`, `df`

A chi-squared random number generator provides different estimates of the standard uncertainty when the degrees-of-freedom `df` is finite.

Example:

```
>>> gen = sim.gaussian_est(1,.5,10)
>>> gen.next()
simulated_input(x=1.7995, u=0.5191, df=10)
>>> gen.next()
simulated_input(x=0.6995, u=0.6150, df=10)
```

uniform_est (*x*, *a*, *df=inf*)

Return a generator ofa uniformly distributed input estimates

Parameters

- **x** – best estimate, used as the mean for simulations
- **a** – half-width, used to calculate a standard deviation for simulations
- **df** – degrees-of-freedom

The generator produces namedtuples with attributes `x`, `u`, `df`

The mean of the simulated distribution is `x` and the standard deviation `u = a/sqrt(3)`.

A chi-squared random number generator provides different estimates of the standard uncertainty when the degrees-of-freedom `df` is finite.

Example:

```
>>> gen = sim.uniform_est(1,.5,10)
>>> gen.next()
simulated_input(x=0.7496, u=0.2880, df=10)
>>> gen.next()
simulated_input(x=0.7080, u=0.3367, df=10)
```

triangular_est (*x*, *a*, *df=inf*)

Return a generator of triangular input estimates

Parameters

- **x** – best estimate, used as the mean for simulations
- **a** – half-width, used to calculate a standard deviation for simulations
- **df** – degrees-of-freedom

The generator produces namedtuples with attributes `x`, `u`, `df`

The mean of the simulated distribution is `x` and the standard deviation `u = a/sqrt(6)`.

A chi-squared random number generator provides different estimates of the standard uncertainty when the degrees-of-freedom `df` is finite.

Example:

```
>>> gen = sim.triangular_est(1,.5,10)
>>> gen.next()
simulated_input(x=0.8494, u=0.1831, df=10)
```

```
>>> gen.next()
simulated_input(x=1.4484, u=0.1783, df=10)
```

arcsine_est (*x*, *a*, *df=inf*)

Return a generator of arcsine input estimates

Parameters

- **x** – best estimate, used as the mean for simulations
- **a** – half-width, used to calculate a standard deviation for simulations
- **df** – degrees-of-freedom

The generator produces namedtuples with attributes `x`, `u`, `df`

The mean of the simulated distribution is `x` and the standard deviation `u = a/sqrt(2)`.

A chi-squared random number generator provides different estimates of the standard uncertainty when the degrees-of-freedom `df` is finite.

Example:

```
>>> gen = sim.arcsine_est(1,.5,10)
>>> gen.next()
simulated_input(x=1.171, u=0.317, df=10)
>>> gen.next()
simulated_input(x=1.236, u=0.437, df=10)
```

mv_gaussian_err (*x*, *cv*)

Return a generator of multivariate Gaussian errors

Parameters

- **x** – the mean
- **cv** – the variance-covariance matrix

Example:

```
>>> z = (1.0,0.5)
>>> cv = la.array( [[0.4,0],[0,0.2]] )
>>> gen = sim.mv_gaussian_err(z,cv)
>>> gen.next()
[1.4171563857707223, 0.8272576411959041]
>>> gen.next()
[1.6483724628734273, 0.2507518428675501]
```

bi_gaussian_err (*z*, *cv*)

Return a generator of bivariate Gaussian errors

Parameters

- **z** – the mean
- **cv** – the covariance matrix

Example:

```
>>> z = 1+0.5j
>>> cv = la.array( [0.4,0,0,0.2] )
>>> gen = sim.bi_gaussian_err(z,cv)
>>> gen.next()
(0.4011898661840365+0.2985638387342745j)
>>> gen.next()
(0.15916435129849904-0.23052895672783447j)
```

uniform_ring_err(*z*, *r*)
Return a generator of random points on a circle

The generator function produces random points distributed on a circle of radius `r`, centre `z`, in the complex plane.

Example:

```
>>> z,r = (0.34-6j), 1.5
>>> rv = sim.uniform_ring_err(z,r)
>>> rv.next()
(1.7951618239337341-6.3640110797295169j)
```

uniform_disk_err(*z*, *r*)
Return a generator of points in and on a disk

The generator function produces random points distributed uniformly on a disk of radius `r`, centre `z`, in the complex plane.

Example:

```
>>> z,r = (12.34-6j), 5.5
>>> rv = sim.uniform_disk_err(z,r)
>>> rv.next()
(13.680838228974434-1.7635650558700604j)
```

mv_gaussian_est(*x*, *cv*, *df=inf*)
Return a generator of multivariate Gaussian input estimates

Parameters

- **x** – best estimate, used as the mean for simulations
- **cv** – standard variance-covariance matrix
- **df** – degrees-of-freedom

The generator produces `simulated_vector_input` namedtuples with attributes `x`, `cv`, `df`.

A multivariate Gaussian distribution is used to to simulate the observations `x`.

A Wishart random number generator is used to provide different covariance matrices when the degrees-of-freedom `df` is finite.

Example:

```
>>> z = (1.0,0.5)
>>> cv = la.array( [[0.4,0],[0,0.2]] )
>>> gen = sim.mv_gaussian_est(z,cv)
>>> gen.next().x
[0.7746525731799878, 0.926462098877803]
>>> gen.next().cv
array([[0.4, 0],
[0, 0.2]])
>>> gen.next()
simulated_vector_input(
    x=[0.4064169945628521, 0.1324584259774531],
    cv=array([[0.4, 0],[0, 0.2]]),
    df=inf)
```

bi_gaussian_est(*z*, *cv*, *df=inf*)
Return a generator of bivariate Gaussian input estimates

Parameters

- **z** – best estimate, used as the mean for simulations
- **cv** – a 4-element sequence representing the covariance

- **df** – degrees-of-freedom

The generator produces `simulated_complex_input` namedtuples, with attributes `z`, `cv`, `df`.

A bivariate Gaussian distribution is used to to simulate observations `z`.

A Wishart random number generator is used to provide different covariance matrices when the degrees-of-freedom `df` is finite.

Example:

```
>>> z = 1+0.5j
>>> cv = la.array( [0.4,0,0,0.2] )
>>> gen = sim.bi_gaussian_est(z,cv)
>>> gen.next().z
(1.1791619306909931+0.56658874952748051j)
>>> gen.next().cv
(0.40, 0, 0, 0.20)
>>> gen.next()
simulated_complex_input(
    z=(1.5483928415509403+2.4126670838378645j),
    cv=(0.40, 0, 0, 0.20),
    df=inf
)
```

interval_OK(*measurand*, *x*, *U*)

Return `True` if the measurand lies inside `[x-U,x+U]`

Parameters

- **measurand** – the true value (float)
- **x** – the estimate (float)
- **U** – the expanded uncertainty (float)

Example:

```
>>> mu, sd = 1.0, 0.1
>>> k = reporting.k_factor()
>>> _OK = lambda m,rv: sim.interval_OK(m,rv.x,k*rv.u)
>>> rv = sim.gaussian_est(mu,sd)
>>> N = 10000
>>> success = sum(
...     _OK(mu, rv.next() ) for i in xrange(N)
... )
>>> success
9523
```

circle_OK(*measurand*, *z*, *U*)

`True` if `measurand` is inside a circle

Parameters

- **measurand** – the true value (complex)
- **z** – the estimate (complex)
- **U** – radius of the expanded uncertainty circle (float)

The circle around `z` with radius `U` is an uncertainty statement for an estimate of the measurand.

Example:

```
>>> k2_sq = reporting.k2_factor_sq()
>>> def _OK(m,rv):
...     U = math.sqrt(k2_sq * rv.cv[0])
...     return sim.circle_OK(mu,rv.z,U)
```

```
...
>>> mu = 1.0+0.3j
>>> cv = [1.0,0.0,0.0,1.0]
>>> rv = sim.bi_gaussian_est(mu,cv)
>>> N = 10000
>>> success = sum(
...     _OK(mu, rv.next() ) for i in xrange(N)
... )
>>> success
9468
```

ellipse_OK(*measurand*, *z*, *cv*, *k2_sq*, *inverted=False*)

`True` when `measurand` lies inside the uncertainty region

Parameters

- **measurand** – the true value (complex)
- **z** – the estimate (complex)
- **cv** – 4-element sequence representing the covariance matrix, or its inverse
- **k2_sq** – bivariate elliptical coverage factor squared (float)
- **inverted** – `True` if *cv* is inverted (Boolean)

The shape of the elliptical uncertainty region around `z` is determined by the covariance matrix `cv`.

When the Mahalanobis Distance squared between *measurand* and `z` is less than the two-dimensional coverage factor squared `k2_sq`, the measurand is within the region.

When `inverted == True`, the inverse of `cv` is not calculated.

Example:

```
>>> k2_sq = reporting.k2_factor_sq()
>>> def _OK(m,rv):
...     return sim.ellipse_OK(mu,rv.z,rv.cv,k2_sq)
...
>>> mu = 1.0+0.3j
>>> cv = [1.0,0.0,0.0,1.0]
>>> rv = sim.bi_gaussian_est(mu,cv)
>>> N = 10000
>>> success = sum(
...     _OK(mu, rv.next() ) for i in xrange(N)
... )
>>> success
9496
```

success_rate_sd(*N*, *p=95*)

Return the standard deviation of the expected number of successes

Parameters

- **N** – the number of trials (integer)
- **p** – the nominal coverage probability (in %)

Assuming a binomial process with probability `p` % of success, when `N` trials are carried out the standard deviation is `sqrt( N*p*(1-p) )`.

Example:

```
>>> sim.success_rate_sd(1000)
6.8920243760451143
```

CHAPTER

NINE

OTHER TOPICS

9.1 Windows command prompt syntax

9.1.1 Interactive mode

GTC can be started in interactive mode by typing GTC at the command prompt.

To stop the interpreter, type quit(), or CTRL-Z.

Script processing

The syntax when running GTC at the command prompt is:

```
GTC [-i | --interact] [drive:][path][file]
```

where

- [ ] indicates an optional element
- -i or --interact : cause GTC to remain in interactive mode after processing input file(s)
- [drive:][path][file] : the drive, path and filename(s) of script files (including file extension)

When there are no command line arguments, the calculator starts an interactive interpreter.

Files passed to the calculator will be executed in the order that they appear on the command line.

Other options

GTC has several other command line options:

```
--version       show the version number
-h, --help      show a brief message about command-line options
-p, --plain     suppress the ``GTC`` banner output at the start of interactive mode
```

9.2 Windows environment variables

9.2.1 The Windows user environment variable PATH

During installation, the location of the folder containing `gtc.exe` is added to the Windows environment variable for the *user's* `PATH`.

9.2.2 The user's environment variable GTC_LIB

During installation, an environment variable `GTC_LIB` is created *for the current user* (if it does not exist already).

The paths in `GTC_LIB` identify folders that `GTC` will search when an `import` statement is executed.

The `lib` folder in the installation directory (e.g.: `C:\Users\user.name\AppData\Local\gtc\lib`) is added to `GTC_LIB` during installation, as is the user's `My GTC\lib` folder.

If other paths are added to `GTC_LIB`, they will not be removed when `GTC` is uninstalled, so software updates will not affect user-defined extensions.

9.2.3 The user's environment variable GTC_SCRIPTS

During installation, there is an option to create a `My GTC` folder in the user's home directory. When this option is selected (by default) an environment variable `GTC_SCRIPTS` is created *for the current user* (if it does not exist already). A folder `My GTC\scripts` is created and the path is added to `GTC_SCRIPTS`.

The `GTC_SCRIPTS` variable is used to identify folders of user defined scripts. When `GTC` runs, it first tries to find a script in the current folder, but if this fails it will search the folders in the order that they appear in `GTC_SCRIPTS`.

9.2.4 Extension modules and packages

Python uses structures called a *module* and a *package*.

- a **module** is a file containing Python definitions and statements
- a **package** is a collection of modules in a folder or folders.

If a `GTC` script contains an `import` statement of the form:

```
import my_module
```

the paths in the environment variable `GTC_LIB`, plus the current working directory, will be searched for `my_module.py` (in addition to a search of the modules included with `GTC`).

If a `GTC` script contains an `import` statement of the form:

```
import my_package.my_module
```

or

```
from my_package import my_module
```

a search for the folder `my_package` is carried out by looking in the paths defined in `GTC_LIB`. When found, `my_package` is then searched for `my_module.py`.

The Python documentation should be consulted for more information about modules and packages.

9.3 Some comments about GTC regression functions

9.3.1 Overview

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

Functions in *type_a* implement a variety of regression 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 *function*, 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, but uncertainties are propagated through the regression equations. The residuals are not used.

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

9.3.2 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 generally correlated.

Weighted least-squares

`type_a.line_fit_wls` implements a so-called weighted least-squares straight-line regression. This assumes that a sequence of uncertainties provided with the input data are known, exactly (i.e., with infinite degrees of freedom). The uncertainty in the slope and intercept is 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.

9.3.3 The `function` module regression functions

Ordinary least-squares

`function.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

`function.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).

[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)

Note: *type_a.line_fit_wls* and *function.line_fit_wls* will 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 *function* module. Using a sequence of u_y values with *function.line_fit_wls* will perform the required calculation.

Weighted total least-squares

function.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 with *function.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).

9.4 Change History

9.4.1 Version 0.9.10

New functions and classes

- The class *type_a.BiasedIndication* as been added. This can be used to correct simple instrument indications for bias using a type-A estimate of the correction term required. In statistical terminology, the uncertain number represents a 'prediction interval' for a future indication.

Changes to existing functions and classes

- A flag `dependent` is provided when calling *core.ureal* and *core.ucomplex*. This should be set `True` if the uncertain number may be correlated with others.
- *core.set_correlation* will issue a warning if used with uncertain number arguments that were not created with the `dependent` flag set `True`. In future releases, this will be a requirement.
- *core.result* is an alias for `archive:result`.
- *reporting.sensitivity* is deprecated. This function will not be available in future and there is no equivalent functionality.
- *core.component* and *reporting.u_component* issue warnings when asked for a component of uncertainty with respect to an uncertain number that is not elementary and has not been declared using *core.result*.
- Uncertain numbers can now be used in Boolean expressions. Uncertain numbers are `True` when the value is non-zero
- A bug in the function *linear_algebra.transpose* and the associated object attribute `T` has been corrected.
- The function *reporting.budget* now returns an empty sequence if the object passed to it has no components of uncertainty (previously a `RuntimeError` was raised).
- A bug has been corrected in the evaluation of degrees of freedom for complex quantities.

9.4.2 Version 0.9.9

This release is mainly concerned with fixing issues.

- Windows Administrator permission is not needed to install `GTC`.
- The installation process can create a `My GTC` folder under the users home folder (e.g., `C:\Users\jane.user\My GTC`). This folder contains an `examples` folder, a `lib` folder and a `scripts` folder. The `lib` folder path is added to the environment variable `GTC_LIB`. The `scripts` folder path is added to the environment variable `GTC_SCRIPTS`. The folders in `GTC_SCRIPTS` will be searched by `GTC` to locate script files.
- The method `archive.Archive.intermediates`, introduced in 0.9.8, has been removed (however, archives created using GTC version 0.9.8 can still be read in 0.9.9).
- The function `archive.result` is now used to declare intermediate results that can be archived.
- *reporting.k_factor* and *reporting.k2_factor_sq* now have quicker, but less accurate, implementations available as an option.
- *core.multiple_ureal* and *core.multiple_ucomplex* can now handle some elements that have zero uncertainty (i.e., by creating constant uncertain numbers)
- Fixed a problem with *function.line_fit*, which did not work properly when the `x` data contained integers (truncation during integer division would occur)
- Fixed a problem with *function.line_fit_wtls*, which did not converge to a solution in certain cases (this bug also affected *type_a.line_fit_wtls*)
- Fixed a problem with *reporting.k_factor*, which was unstable for very large values of `df`
- The Windows Explorer context menu 'SendTo' `GTC` item now passes the target folder to `GTC`, allowing the script to import modules in that folder.
- *function.mean* now accepts an iterable object. Previously a sequence was required, but it now accepts generator objects as well.

9.4.3 Version 0.9.8

This release is mainly concerned with fixing problems identified in the previous versions.

In particular, the *Archive* class has been redesigned.

When an archive is created, the default behaviour now ignores the relationships between any archived intermediate uncertain numbers. When it is necessary to save information about these relationships, the new method `intermediates` should be used to identify those intermediate uncertain numbers of interest.

Note: Archives created using GTC version 0.9.7 are incompatible with 0.9.8. The older format archives need to be recreated using the new version.

- The core function *get_covariance* has been added
- Uncertain real numbers now have the attributes *.real* and *.imag*
- GTC `import` now finds modules located in the current working directory
- The sign of the angle returned by *u_rect_to_tangent* has been corrected
- *mag_squared* now handles uncertain real numbers correctly
- Complex division when the numerator value was zero has been fixed
- *get_correlation* now returns zero, or zero seq, when real or complex numbers are given
- The degrees of freedom calculation for the imaginary component of an uncertain complex number has been fixed

9.4.4 Version 0.9.7

New functions and classes

- *function*
  - *line_fit_wls*
  - *mean*
  - `intermediate`
  - *mul2*
- *type_a*
  - *line_fit_wls*
  - *line_fit_rwls*
  - *merge_components*
- *archive*
  - *Archive*
  - *load*
  - *dump*
  - *loads*
  - *dumps*
- *reporting*
  - *sensitivity*
  - *k_to_dof*

- *k2_to_dof*

Changes to existing functions and classes

- *function.line_fit*

  The function signature has changed. The optional *a_b* parameter has been removed.

- *function.line_fit_wtls*

  Name changed from `line_fit_TLS`.

  The function signature has changed. The initial estimates of slope and intercept are no longer the first argument.

- *function.LineFitOLS*, *function.LineFitWLS* and *function.LineFitWTLS*

  These are refinements of the `LineFit` in the previous release. They are specific to the type of regression that has been performed.

- *type_a.line_fit*

  A label can be assigned to the uncertain number parameters *a* and *b*.

  A different type of object is returned.

- *type_a.line_fit_wtls*

  Name changed from `line_fit_TLS`.

  The function signature has changed. The initial estimates of slope and intercept are not the first argument.

  A label can be assigned to the uncertain number parameters *a* and *b*.

  A different type of object is returned.

- *type_a.LineFitOLS*, *type_a.LineFitWLS*, *type_a.LineFitRWLS*

  These are refinements of `LineFit`. They are specific to the type of regression that has been performed.

  The `chi_square` attribute is now called `ssr` and the attribute `N` has been added.

- *type_a.LineFitWTLS*

  As above for *type_a.LineFitOLS* and *type_a.LineFitWLS*.

  This class has additional functions `x_from_y` and `y_from_x`. These, can be used to predict the independent variable from observations of the dependent variable and predict the dependent variable from a value of the independent variable.

- *type_a.mean*

  Although the elements of the data sequence may be uncertain numbers, only the value attribute is used to calculate the mean and only a pure number (float or complex) is returned.

  To evaluate the uncertain-number mean of a sequence use *function.mean*.

- *sim*

  A number of changes have been made to the `sim` module.

  A suffix has been added to the functions `gaussian`, `uniform`, `triangular` and `arcsine`. They are now *gaussian_est*, *uniform_est*, *triangular_est* and *arcsine_est*.

  A suffix has also been added to `bi_gaussian` and `mv_gaussian`. These functions are now *bi_gaussian_est* and *mv_gaussian_est*.

  The mapping `distribution` has been replaced by a mapping `estimate` in which the names (without the `_est` suffix) are keys to the functions above.

A suffix has been added to `uniform_ring` and `uniform_disk`. These are now named *uniform_ring_err* and *uniform_disk_err*.

New functions `gaussian_err`, `uniform_err`, `triangular_err`, `arcsine_err`, *bi_gaussian_err* and *mv_gaussian_err* have been added.

A mapping `error` is provided in which the names (without the `_err` suffix) are keys to the functions above.

9.4.5 Version 0.9.6

Changes affecting the GTC interpreter

- The standard Python modules `math` and `cmath` are now imported into the `GTC` environment. So there is no need to use `import math` or `import cmath`, unless writing extension modules.
- Integer division no longer truncates a quotient that is not exactly an integer. So now

```
>>> 2/3
0.6666666666666666
```

New functions

New functions have been added to modules: *core*, *function*, *type_a*, *reporting* and *number_strings*.

- *core*
  - *multiple_ureal*
  - *multiple_ucomplex*
- *function*
  - *line_fit*
  - line_fit_TLS
- *type_a*
  - *standard_deviation*
  - *standard_uncertainty*

    the previously separate functions `standard_uncertainty_real` and `standard_uncertainty_complex` have been combined
  - *estimate*

    now returns an uncertain number, instead of a tuple of sample statistics
  - *multi_estimate_real*
  - *multi_estimate_complex*
  - *estimate_digitized*
  - *line_fit*
  - line_fit_TLS
  - *chisq_p*
  - *chisq_q*
- *reporting*
  - *u_bar*

name changed: was called `fn_bar`

- *v_bar*

  name changed: was called `tv_bar`

- *budget*

  added an option to display only the more significant uncertainty components

- *number_strings*
  - *open*
  - *File*

    The *open* creates a *File* object that converts numbers to and from their text representation without loss of internal precision.

Changes affecting user-defined modules

- User-defined extension modules can now use the Python option

```
from __future__ import division
```

  to ensure that integer division no longer truncates a quotient that is not exactly an integer.

9.4.6 Version 0.9.5

The first release of `GTC`.

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z