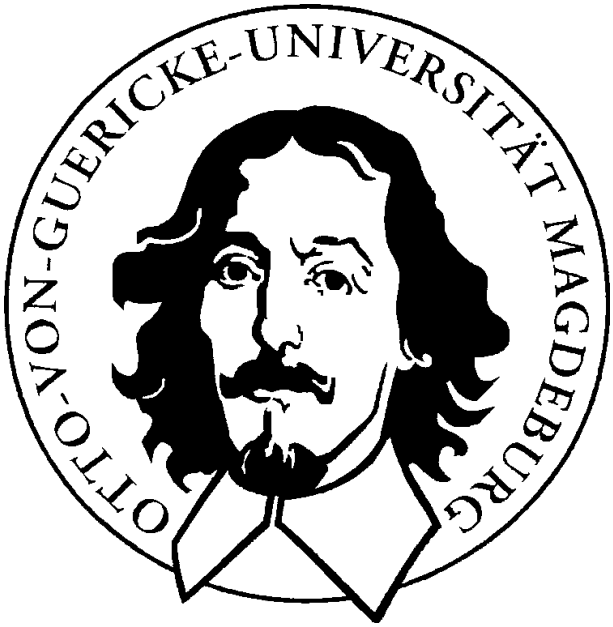
SCUQ – A Class Library for the Evaluation of Scalar- and Complex-Valued Uncertain Quantities

Thomas Reidemeister



A thesis submitted to the Otto-von-Guericke University, Magdeburg
towards the degree of Diplom Ingenieurinformatiker (Dipl. Ing.-Inf.).

Themensteller:	Dr. rer. nat. Hans Georg Krauthäuser
Betreuer:	Prof. Dr. rer. nat. habil. Jürgen Nitsch
	Prof. Dr.-Ing. habil. Georg Paul
Abgabetermin:	18.02.2007
vorgelegt von:	Thomas Reidemeister Lemsdorfer Weg 22 39112 Magdeburg treideme@student.uni-magdeburg.de

Declaration / Erklärung

This thesis is the result of my own work and includes nothing that is the outcome of work done in collaboration. I hereby certify that I did not use any other references than those stated and that direct or indirect ideas taken over from elsewhere are marked as such.

Ich versichere hiermit, dass ich die vorliegende Diplomarbeit selbständig, ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Abstract

A measurement evaluation is not complete, unless the uncertainty and unit of the measured quantity are also reported in addition to the measured value. In general, the measured quantity is based on several correlated other quantities. In most cases the physical model relating these input parameters is known. A standard evaluating the uncertainty of the measured quantity in this case is proposed by the ISO Guide to the expression of uncertainty in measurements (GUM) [1]. The GUM proposes using the means of frequential statistics to evaluate the problem. Therefore effects that cannot be expressed using relative long run frequencies (*i.e.* repeated measurements) are randomised. Although the GUM is widely accepted and has been ported to national standards (*e.g.* DIN [2], and NIST [3]), it is a proven approximation of systematic effects contributing to the uncertainty of the measured quantity.

Weise and Wöger [4] propose an alternate approach using Bayesian inference. Bayesian inference allows incorporating a degree of belief into the statistical evaluation of a quantity. Therefore, their method can model systematic effects more accurately than the GUM does. According to the authors, the outcomes of their method are in line with the requirements of the GUM.

In this thesis, we evaluate both approaches in general and evaluate the following problems in particular:

- propagation of the uncertainty of scalar-valued (real) input quantities contributing to one scalar (real) output parameter,
- propagation of the uncertainty of complex-valued input quantities contributing to one complex-valued output parameter.

Related work focusing on these problem domains is reviewed and a class library implemented in Python [5] that can be used by software applications to propagate the uncertainty in both cases automatically, is presented. We implemented methods that use frequential statistics and present a software design implementing the method proposed by Weise and Wöger. Furthermore, we implemented classes evaluating the physical units of all quantities of the physical model. These classes can be used optionally to verify the physical model.

We tested the class library by evaluating solved problems of other researchers. Furthermore, we verified our implementation using a suite of component tests that can also be used to test the compatibility of the destination platform. Our library evaluated all presented problems correctly, and it was tested using Python 2.4 and NumPy 1.0.1 (*see* Oliphant [6]) on Microsoft Windows XP and SUSE Linux 9.3 on 80x86 platforms.

We are confident that this class library can be used as a cornerstone in many software applications for uncertainty propagation.

Contents

List of Figures

List of Tables

1. Introduction

A measurement is the estimation of a physical quantity. The uncertainty of a measurement quantifies the trust that the estimate expresses the true value of the measured quantity. In most cases the desired quantity is not measured directly. Instead it is estimated based on measurements of other influence quantities that contribute to the measured quantity. We refer to a method evaluating the uncertainty of a measured quantity based on influence quantities as propagation of uncertainty.

In the context of electrical measurements, the measured signals are often not only isolated, decoupled, and transmitted, they are also amplified, compensated, transformed, filtered, converted, and linearized, before an estimate of the true value is printed on the screen of the measuring device (*see* Schrüfer [8]). This example illustrates two properties of a measurement. First, it is impossible to present the true value of the measured quantity. Second, the estimated quantity is based on a model that can only approximate many factors that contribute to the uncertainty of the measurement. Kessel [9] exemplifies several effects contributing to the uncertainty of a measurement, shown in Table 1.

Random Effects	Systematic Effects
Temperature fluctuations	Drift of the measuring device (*i.e.* due to aging)
Noise	Uncertainty of the reference quantity
Irregular performance of the observer	A biased observer

Table 1: Some examples of random and systematic effects

Schrüfer distinguishes between two major sources of effects that contribute to the uncertainty: random effects and systematic effects. Random effects arise from non-ascertainable fluctuations of the measuring device. Systematic effects are known.

There are many important areas, in which the expression of uncertainty is crucial. Examples are calibration, compliance with quality and safety standards, and the comparison of measurements.

Unfortunately, the importance of a proper uncertainty evaluation is sometimes underestimated for various reasons, which may lead to consequential damage; for example, Oberg [10] describes the reasons why the NASA Mars Climate Orbiter was destroyed in 1999. Two of the reasons contributing to the doom of the Mars probe were a poor evaluation of the navigation uncertainty and the confusion of different unit systems. Finally, the $125,000,000$ USD Climate Orbiter "[...] metaphorically speaking, marched off the cliff and was destroyed [...]" (Oberg [10]).

Other reasons for a poor evaluation of the uncertainty in general maybe the complexity of the propagation of uncertainty and the lack of proper software support.

There are several proprietary software tools available such as the GUM workbench 1.3 [11]. It does not integrate into the measurement evaluation directly. The data has to be imported by the user or has to be obtained using other proprietary products. It supports physical units; however, they are just checked symbolically. Although the GUM workbench was validated by the PTB (*see* PTB note [12]), the internals remain undocumented and the source code is unavailable. Furthermore, the creators of the GUM workbench require the users of the software to submit to a restrictive software license.

Another approach automating the evaluation of uncertainty is using class libraries such as ByGUM [7]. The class libraries better integrate into existing programs that evaluate the mea-

sured quantity compared to standalone software solutions. ByGUM is implemented using an interpreted language; therefore, it can also be used interactively. It does not provide support for physical units. ByGUM is only freely available for non-commercial use and requires the users to register their software. We are not aware of a class library for the propagation of uncertainty that integrates physical units.

The goal of this thesis is to present our class library SCUQ. SCUQ is the abbreviation for "class library for the automatic evaluation of **S**calar or **C**omplex-valued **U**ncertain **Q**uantities". It assists in the propagation of uncertainty in measurements in general and in the context of electrical measurements in particular. We have chosen to implement the Gaussian error propagation law for scalar and complex-valued influence quantities. We define *scalar quantities* being quantities that are expressed using real values. The use of the Gaussian error propagation law is proposed by the GUM and is therefore widely accepted. However, it is an approximation because it linearizes the model in the region of the estimated value. In order to solve this issue, we also provide a software design using Bayesian inference allowing a more accurate representation of systematic and random effects than the Gaussian error propagation law does. Unfortunately, the method is very computation-intensive, which limits its domain of application. Furthermore, we evaluate the methods in this thesis leaving the choice to the user, which method suits their application best. In addition, our class library supports physical quantities such that units can be assigned to numeric values and uncertain quantities.

In Section 2 we describe the context, in which our class library is embedded in. In the Sections 3, 4, and 5 we describe the methods of uncertainty propagation, and our software designs realizing them. Furthermore, we present and assess the approaches integrating units in software in Section 6. Finally, we demonstrate the usage of the class library in Section 7 and conclude the lessons learned in Section 8. The programming manual for SCUQ is included in Appendix C. It describes the interfaces of all classes and demonstrates the usage of SCUQ.

2. The Software Environment

In this section we provide an overview of the software environment SCUQ is embedded in. SCUQ stands for *class library for the automatic evaluation of **S**calar or **C**omplex-valued **U**ncertain **Q**uantities*. We implemented it in the Python programming language (*see* [5]) using NumPy (*see* [6]). The classes use native Python data types and functions as well as data types and functions of NumPy. NumPy is a third party collection of types and functions supporting scientific computing. We also implemented an interface for NumPy. Figure 1 shows the components and their interfaces within the environment of SCUQ.

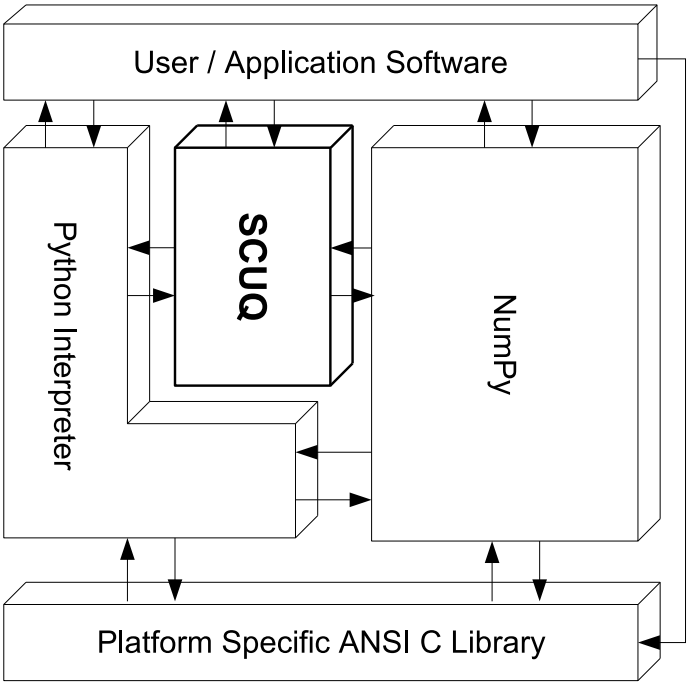


Figure 1: The software environment of SCUQ

Since Python is an interpreted language, our classes can also be used interactively in the Python interpreter and related applications. We developed an interface for NumPy so that applications using NumPy can integrate our class library. NumPy as well as the Python interpreter provide APIs for ANSI C. This allows the integration of SCUQ in applications using these APIs.

Since we implemented our design using NumPy and Python, the desired hardware platforms are those that support both applications. Since Pythons functionality as well as NumPy functionality depend on the underlying C-library we cannot guarantee the proper operation of SCUQ on all platforms. We implemented a test-suite consisting of several test cases that verify the proper operation of SCUQ (*see* Section 2.3).

Section 2.1 presents a short overview of the Python programming language and a description of the components of Python used for our design. Section 2.2 describes NumPy briefly. Furthermore, we describe how SCUQ interacts with NumPy. In Section 2.3, we describe a suite of self tests that is included in SCUQ and the tested platforms.

2.1. The Python Programming Language

The Python programming language is a free platform-independent general-purpose programming language (*see* [5]). According to Gupta [13], Python is the ideal choice for applications that require the features of a scripting language as well as the features of an interpreted language. It evolved from the ABC programming language that was used for teaching purposes in the 1980s.

Gupta [13] identified the following features of Python:

- easy: Python has a compact notation that is similar to Algol, C, and Pascal. It does not have extra symbols for starting and ending code blocks. Instead indentation is used to group statements.
- scalable: Compared to scripting languages of the Unix-environment, Python provides better structure support for large programs. The code can be grouped in modules.
- high level: Python has built-in modules to make system calls. It contains support for storage types, such as lists, arrays and hash-tables, allowing expressing complicated expressions in a single statement.
- object oriented: All components of the Python programming language are objects. Python supports late binding, multiple inheritance, and polymorphism, allowing the creation of object-oriented class hierarchies. Moreover, operators can have different meanings according to the elements being referenced. Python is a dynamic-typed language. Therefore, the type of an object is determined at run time only.
- interpreted: Python supports byte compilation like the Java programming language does. In addition, Python programs can be run, debugged, and tested interactively in the Python interpreter.
- extensible and flexible: Python uses the same interface for built-in and third party modules. Classes can be grouped in modules. Native code can be integrated using a C API.
- rich core library
- memory management: The interpreter manages the memory, thus removing additional programming overhead from the developer.
- exception handling: Python supports exceptions similar to Java ((*see* [14])) and C++.
- portability: The Python interpreter was ported to variety of hard- and software platforms because it is implemented in ANSI C.
- freeware: Python is freely available. The code created using the Python interpreter can be sold or distributed by the creator in any manner.

According to Gupta [13] Pythons main application areas are:

- integrating large software components written in C, C++, and Java,
- prototyping of applications that are about to be implemented in C, C++, or Java,
- writing CGI scripts; this is because of Pythons strong presence on the Web.

In addition to Guptas [13] enumeration of application areas, we identified scientific computing as application area of Python. Because Python provides native support for complex numbers and there are a variety of 3rd-party modules for Python for scientific computing such as NumPy (*see* Oliphant [6]). It supports large-scale numeric arrays and matrices, linear algebra, random number generation, and discrete Fourier transformations. However, NumPy defines own numeric types that have a similar interface like Pythons numeric types.

After presenting the features of Python in general, we describe the Python types we used for our design in detail. This description can be used as guideline for implementing the environment of our design in other object oriented programming languages. In Section 2.1.1 we describe Pythons numeric types. In Section 2.1.2 we describe its storage types.

2.1.1. Numeric Types

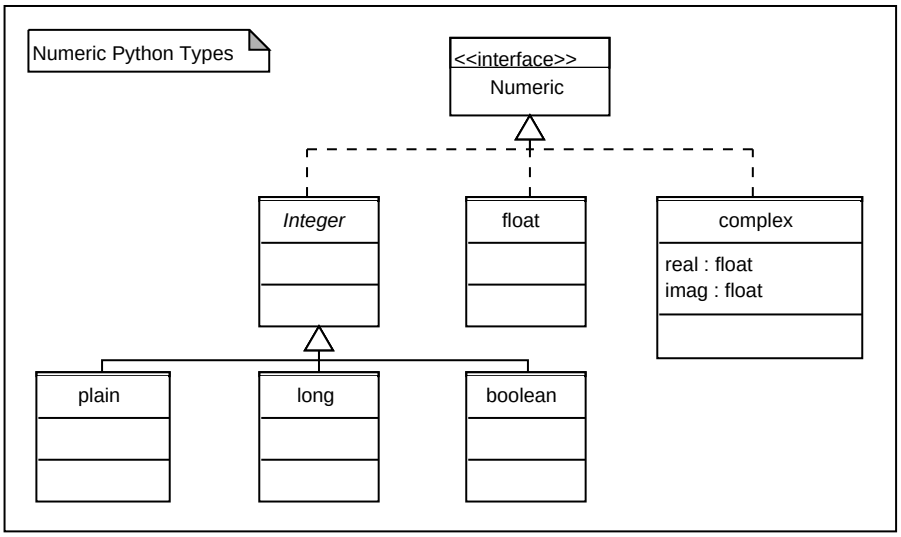


Figure 2: Numeric Python types

Python implements a variety of numeric types. Their hierarchy is shown in Figure 2. The accuracy and range of the built-in types are platform dependent:

- The type `int` implements the type `long int` of the platforms C library having at least 32 bits of precision.
- `long` is a platform independent type modelling integers of arbitrary precision.
- The type `boolean` is just an alias for `int` that is used to define the constants `True` and `False`.
- The type `float` implements floating-point numbers. The precision as well as the capabilities depends on the platforms C library type `double`; for example, Microsoft Visual C++ does not implement the values infinity (`inf`) and not a number (`NaN`).
- The type `complex` implements complex numbers. It is backed up by two floating-point numbers representing the real and the imaginary part respectively.

All types implement the binary operators shown in Table 2 according to the arithmetic of the type. The same applies to the unary operations shown in Table 3. All types except for `complex` can be converted to each other using the operators shown in Table 4. They also implement conversion to other numeric types shown in Table 4. Instances of `complex` raise an error whenever they are about to be converted to another type using these methods.

If a class should model a custom numeric type, no inheritance is needed. Only the methods shown in Table 2 need to be implemented. Therefore we denoted the abstract type `Numeric` as an interface defining the methods shown in the Tables 2, 3, and 4.

Operation	Operator	Method
Addition	+	`__add__(self, other)`
Subtraction	-	`__sub__(self, other)`
Multiplication	*	`__mul__(self, other)`
Division	/	`__div__(self, other)`
Division	//	`__truediv__(self, other)`
Power	**	`__pow__(self, other)`
Shift Left	<<	`__lshift__(self, other)`
Shift Right	>>	`__rshift__(self, other)`
And	&	`__and__(self, other)`
Exclusive Or	\	`__xor__(self, other)`
Or	|	`__or__(self, other)`
Modulo	%	`__mod__(self, other)`
Less Than	<	`__lt__(self, other)`
Less or Equal	<=	`__le__(self, other)`
Equal	==	`__eq__(self, other)`
Not Equal	!=	`__neq__(self, other)`
Greater Than	>	`__gt__(self, other)`
Greater or Equal	>=	`__ge__(self, other)`

Table 2: Binary operators for numeric types in Python

Operation	Operator	Method
Negation	-	`__neg__(self)`
Inversion	~	`__inv__(self)`
Absolute Value	`abs()`	`__abs__(self)`
Non Zero	`bool()`	`__nonzero__(self)`

Table 3: Unary operators for numeric types in Python

Desired Type	Operator	Method
complex	`complex()`	`__complex__(self)`
int	`int()`	`__int__(self)`
long	`long()`	`__long__(self)`
float	`float()`	`__float__(self)`

Table 4: Conversion operators for numeric types in Python

```
1
def rational(n,d = 1):
3   return RationalNumber(n,d)

5 class RationalNumber:

7   def __init__(self,numerator,denominator):
      self.n = long(numerator)
9     self.d = long(denominator)
      self.normalize()
11
    def __mul__(self,value):
13    n = self.n * value.n
      d = self.d * value.d
15    return rational(n,d)

17  def __add__(self,value):
      n = self.n * value.d + self.d * value.n
19    d = self.d * value.d
      return rational(n,d)
21
    def normalize(self):
23    my_gcd = RationalNumber.gcd(abs(self.n), self.d)
      self.n /= my_gcd
25    self.d /= my_gcd

27  def gcd( m, n ):
      if( n == 0L ):
29      return m
      else:
31      return RationalNumber.gcd( n, m % n )
    gcd = staticmethod(gcd)
33
    def __float__(self):
35    return float(self.n)/self.d

37  def __str__(self):
      return "("+str(self.n)+"/"+str(self.d)+")"
39
    def __coerce__(self,value):
41    if(long(value) == value):
        return (self,rational(value))
43    else:
        return (float(self),value)
45
r1 = rational(2,4)  # 2/4
47 print r1          # (1/2)
   print (r1 + 2)    # (5/2)
49 print (r1 + 0.5)  # 1.0
```

Listing 1: Example: custom numeric types

In order to demonstrate the creation of a custom numeric type, we present an implementation of

a rational number type of arbitrary precision in Listing 1. In order to keep the example compact, we limit the implementation to the operations `+`, `*`, and `float()`. In lines 7–10 we define the default constructor. It takes two values representing the nominator and the denominator of the rational number. The arguments are converted to `long` and stored as the members `n` and `d` of the class `RationalNumber`. The operators `+` and `*` are implemented using the methods `__add__` and `__mul__` respectively. The method `normalize` converts the rational number into its canonical form. The member `__float__` converts the instance to an instance of `float`. We demonstrate the use of the type in lines 46–49. At first we define the rational number $\frac{2}{4}$. In the next line we show that the constructor creates the canonical form of it. The following lines demonstrate the operations `*` and `+`. The operations are evaluated in two steps:

1. If the types of the operands differ, Python calls the method `__coerce__` on the left operand. The method should return a pair of variables of the same type.
2. If both arguments have the same type, Python invokes the respective method on the left operand and provides the right operand as argument. If however the arguments are of a different type, step one is repeated or an exception is raised if the coercion is impossible.

Figure 3 illustrates the general evaluation of coercion rules. Lets take a closer look at `__coerce__` shown in lines 40–44. If the argument `value` can be converted to an integer, we create the rational number $\frac{\text{value}}{1}$. Otherwise, the implementation falls back to floating point arithmetic.

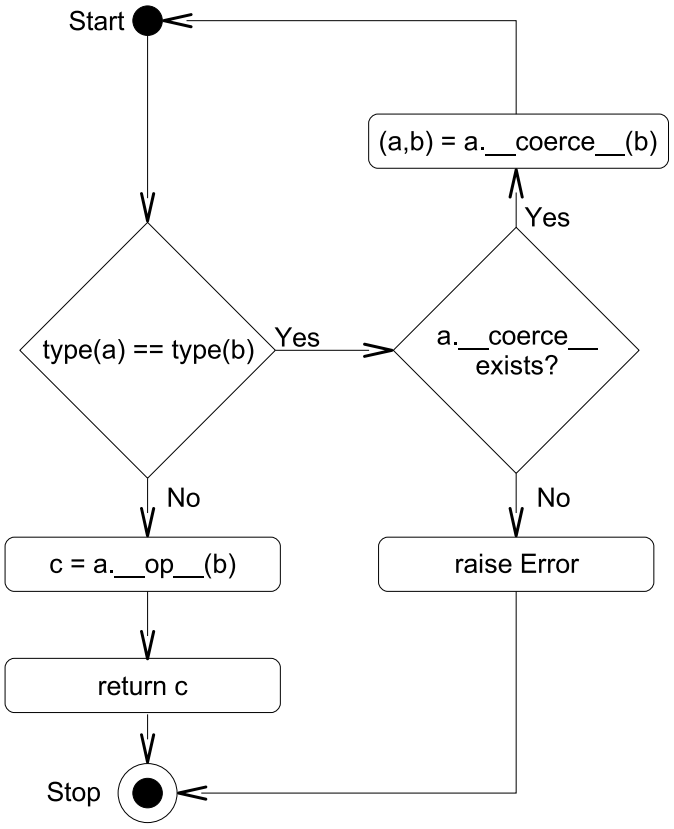


Figure 3: Coercion evaluation in Python

2.1.2. Storage Types

Python defines several storage types as shown in Figure 4.

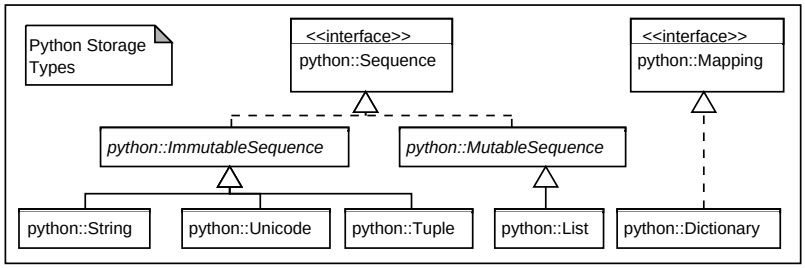


Figure 4: Python storage types

In general, there are two classes: sequences and mappings. Sequences store the elements like an array indexing each element using an unique integer. Mappings act like a hash-table mapping the elements to keys that can be of an arbitrary immutable type.

Operator	Method	Purpose
`len()`	`__len__(self)`	Returns the stored number of elements.
`self[key]`	`__getitem__(self, key)`	Returns the value of the element that is referenced by key.
`self[key]`	`__setitem__(self, key)`	Sets the value of the element that is referenced by key. Dictionaries create a new entry for key, if the key is not already set. For sequences the key must be an integer within the range 0 < len().
`self[key]`	`__delitem__(self, key)`	Deletes the entry that is referenced by key.
`iterkeys()`	`__iter__(self)`	Returns an iterator over the elements of the storage type.
`contains()`	`__contains__(self,item)`	This method tests if an element is contained in the storage type.

Table 5: Methods of storage types

A distinction is drawn between mutable and immutable sequences. Immutable sequences raise an error whenever their content is about to be modified. Sequences and dictionaries differ in the retrieval and placement of their entries. A sequence `seq` in Python contains *N* entries that can be accessed by `seq[i]` and `i < len(seq)`. Dictionaries accept any immutable Python object as argument `i`. They are hash-tables; the argument is the key. Sequence types provide support for slicing. In order to extract a subsequence from element `a` up to element `b` from a sequence `seq`, the expression `seq[a:b]` can be used. If all elements up to the element `a` need to be extracted then `seq[:a]` can be used. All container types implement the operations shown Table 5.

2.2. NumPy

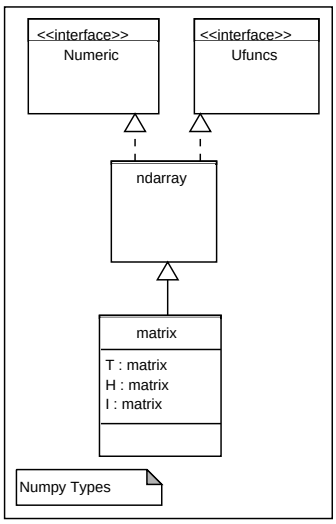


Figure 5: NumPy type hierarchy

After having described some of the Pythons built-in types, we describe the types of NumPy (*see* Oliphant [6]) that were used in our implementation. NumPy [6] is a third-party module for scientific computing that includes classes and functions to solve problems of linear algebra, ordinary differential equations, fast Fourier transformations, and other problems. Its major contribution is an array type that implements the built-in numeric methods of Python and supports so-called *universal broadcasting functions* (ufuncs). These functions are common mathematical functions that operate on real or complex numbers. Used on arrays, they work on each element. The chosen NumPy types hierarchy is shown in Figure 5.

The type `ndarray` supports large n-dimensional arrays. Each array contains only objects of one type; therefore, `ndarray` is also referred to as *homogenous array*. It implements the ufuncs, the standard methods for numeric types in Python, and the operations for Python storage types.

Function	Description
`arccos(x)`	Returns the inverse cosine of the parameter `x`.
`arccosh(x)`	Returns the inverse hyperbolic cosine of the parameter `x`.
`arcsin(x)`	Returns the inverse sine of the parameter `x`.
`arcsinh(x)`	Returns the inverse hyperbolic cosine of the parameter `x`.
`arctan(x)`	Returns the inverse tangent of the parameter `x`.
`arctan2(x,y)`	Returns the inverse tangent of the operation `x/y`.
`arctanh(x)`	Returns the inverse hyperbolic tangent of the parameter `x`.
`cos(x)`	Returns the cosine of the parameter `x`.
`cosh(x)`	Returns the hyperbolic cosine of the parameter `x`.
`exp(x)`	Returns the exponential value of the parameter `x`.
`hypot(x,y)`	Returns the value of the operation $\sqrt{x^2+y^2}$.
`log(x)`	Returns the Natural logarithm of the parameter `x`.
`log10(x)`	Returns the decadic logarithm of the parameter `x`.
`log2(x)`	Returns the binary logarithm of the parameter `x`.
`tan(x)`	Returns the tangent of the parameter `x`.
`tanh(x)`	Returns the hyperbolic tangent of the parameter `x`.
`sin(x)`	Returns the sine of the parameter `x`.
`sinh(x)`	Returns the hyperbolic sine of the parameter `x`.
`sqrt(x)`	Returns the square-root of the parameter `x`.
`fabs(x)`	This method is an alias for the method `abs(x)`.

Table 6: The NumPy ufuncs we implemented in our design

Table 6 shows the selection of ufuncs used in our design. If one of these functions is called on an `ndarray` or its siblings, it is broadcasted to the individual elements of the array; such that, the stored Python object is called with the respective method and the result of the operation is stored in the return array. If the respective method is not implemented by the array element an error occurs. Hence, not all ufuncs need to be implemented by an object in order to be stored in an `ndarray`. An extension of the type `ndarray` is the type `matrix`. It implements Pythons numeric methods for multiplication and powers to support matrix multiplication and powers of quadratic matrices. Consider the example shown in Listing 2. We implemented the type `pair`.

```
# This line imports the 3rd party module numpy.
from numpy import *

# This class implements our custom type for pairs.
class pair:
  # The default constructor of our pairs class
  def __init__(self, a, b):
    self.a = a
    self.b = b

  # The ufunc that is called for +
  def __add__(self, other):
    new_a = self.a + other.a
    new_b = self.b + other.b
    return pair(new_a, new_b)

  # This method allows the printing of our pairs.
  def __str__(self):
    return "(%s,%s)" % (self.a,self.b)

# generate two arrays of 10 pairs
a_1 = array([pair(a,a+1) for a in range(10)])
a_2 = array([pair(a,a+1) for a in range(10)])

print a_1 + a_2
# Result: [(0,2) (2,4) (4,6) (6,8) (8,10) (10,12) (12,14)
# (14,16) (16,18) (18,20)]
```

Listing 2: Example: using universal broadcasting functions

We assume that an addition of pairs takes place element wise. Such that $(a,b)+(c,d) = (a+c,b+d)$. We create two `ndarray`-instances of ten pairs and add the two `ndarray`-instances. The operator + invokes the arrays method `__add__`, which then broadcasts the operation to the individual pair elements. The result of the addition of the pairs is stored in an temporary array and afterwards returned by the method `__add__` of `ndarray`. Note that we only implemented the method for the ufunc add and not all ufuncs. We also did not inherit any functions from base classes. These are the fundamental properties that distinguish broadcasting from inheritance. Broadcasting is fundamental for integrating our design into NumPy, we later describe in the Sections 3 and 4.

2.3. Supported Platforms

Python [5] is available for a variety of platforms including Windows (*see* Microsoft [15]), Linux (*see* GNU Software Foundation [16]), and Mac OS (*see* Apple [17]). However, each distribution strictly depends on the underlying operating system and its C API, especially with respect to the mathematical functions; for example, the floating point API of Microsoft Windows does not support the values infinity (`inf`) and not a number (`NaN`). Furthermore, the accuracy of the floating point operations may vary among the platforms.

Because of these issues we cannot guarantee that SCUQ is fully operational on all platforms supported by Python and NumPy. In order to exclude the platforms, on which our implementation is *not operational*, we developed a suite of test cases. It is defined in the module `scuq.testcases` and is invoked using `python scuq/testcases.py`. We defined component tests for the classes of SCUQ and also test their interaction using the examples described by Hall in [7], [18], and [19]. We compare the outcomes stated in the publications to the outcomes of our component tests with an absolute accuracy of 10^{-6}. That said, passing all tests is no guarantee for SCUQ to be operational, since we assume that Pythons floating point arithmetic is working correctly. The console output for passing all tests is shown in Listing 3.

```
1  workspace/Prototyp> python scuq/testcases.py

3  ...
   ! @brief Test the integration of quantities of the Module cucomponents
     . ... ok
5  ! @brief Test instances of cucomponents.Sin. ... ok
   ! @brief Test instances of cucomponents.Sinh. ... ok
7  ! @brief Test instances of cucomponents.Sqrt. ... ok
   ! @brief Test instances of cucomponents.Sub. ... ok
9  ! @brief Test instances of cucomponents.Tan. ... ok
   ! @brief Test instances of cucomponents.Tanh. ... ok
11
   ----------------------------------------------------------------------
13 Ran 135 tests in 0.755s

15 OK

17 workspace/Prototyp>
```

Listing 3: Console ouput of successful tests

If however a single or more tests fail, it is most likely that SCUQ is not operational on the respective platform. The description of the individual tests is included in the programming manual (*see* Appendix C). Table 7 contains the platforms, on which SCUQ was tested successfully.

Architecture	Operating System	Python Interpreter	NumPy
Intel Xeon 3.06 4 GB RAM	SUSE Linux 9.3 Kernel: 2.6.11.4-21-bigsmp	Python 2.4 built using GCC 3.3.5	NumPy 1.0.1 built using GCC 3.3.5
Intel Xeon 3.06 4 GB RAM	Microsoft Windows XP Professional 2002 Service Pack 2	Python 2.4.3 built using Microsoft Visual C++ v.1310	NumPy 1.0.1 Binary dist. from `www.scipy.org`
Intel Celeron M 256 MB RAM	SUSE Linux 9.3 Kernel: 2.6.11.4-21-default	Python 2.4 built using GCC 3.3.5	NumPy 1.0.1 built using GCC 3.3.5

Table 7: Tested platforms

Based on the evaluation of the platforms described above the following issues are known:

- The tests of the rational number type fails when using NumPy 1.0.1 RC. This issue can be solved by upgrading to NumPy 1.0.1 (release). Otherwise the rational numbers cannot be used in combination with the ufuncs `max` and `min`.
- The values `inf` and `NaN` are not available on Windows. This problem can be avoided by migrating to another platform such as Linux.

3. Propagation of Uncertainty of Scalar Quantities

In this section we evaluate the problem of the propagation of uncertainty of scalar quantities and review the approach proposed by the Guide to the Propagation of Uncertainty in Measurements (GUM) [3] in Section 3.2. In Section 3.3, we present a software design evaluating the uncertainty propagation, based Hall's proposals [20] and [19].

3.1. The Problem

Suppose several estimated scalar input quantities $(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n)$ are related by a physical model, shown in Equation 1. The uncertainty $u(x_i)$ of each input quantity x_i is known, and the input quantities may be correlated. The input quantities are also referred to as *components of uncertainty*.

$$y = f(x_1, x_2, \ldots, x_n) \tag{1}$$

The problem is the evaluation of the uncertainty $u_c(y)$ of the model output y. The uncertainty of y is also referred to as *combined standard uncertainty* that quantifies a symmetric interval $[\hat{y} \pm u_c(y)]$, in which the true value y is supposed to be with a level of confidence of approximately 68%. In addition, it is sometimes required to provide a custom confidence interval $[\hat{y} \pm k u_c(y)]$ for the estimated value of y.

3.2. The GUM Approach

Taylor *et al.* [3] describe a standard for the uncertainty evaluation in various areas (e.g. basic and applied research, international comparisons of measurement standards, generation of standard reference data). Their approach is based on separating the components of uncertainty into two groups: *components of uncertainty arising from random effects* (type A) and *components of uncertainty arising from other effects* (type B). These components can therefore be evaluated by a *statistical analysis* (type A evaluation) or by *other means* (type B evaluation). The result of either type A or type B evaluations is presented as a standard deviation. They refer to it as standard uncertainty. A type B evaluation is based on background knowledge about the respective quantity (*i.e.* knowledge of systematic effects).

After having described the evaluation of the input quantities, we describe the computation of the combined standard uncertainty. The GUM [3] recommends using the Gaussian error propagation law shown in Equation 2.

$$u_c^2(y) = \sum_{i=1}^{N} \left(\frac{\partial f}{\partial x_i} \right)^2 u^2(x_i) + 2 \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{\partial f}{\partial x_i} \frac{\partial f}{\partial x_j} u(x_i, x_j)^2 \tag{2}$$

The Gaussian error propagation law linearizes the model function in the region of y in order to evaluate its uncertainty $u_c(y)$. The input parameters are the following.

- $u(x_i)$: the uncertainty of the input parameter x_i,
- $u(x_i, x_j)$: the estimate of the covariance of the input quantities x_i and x_j,
- $\frac{\partial f}{\partial x_i}$: the partial derivate of the model function with respect to x_i.

The output is $u_c(y)$, the estimated combined standard uncertainty. It is in general only an approximate solution because it is based on the first-order Taylor series of the model (*see* Equation 1).

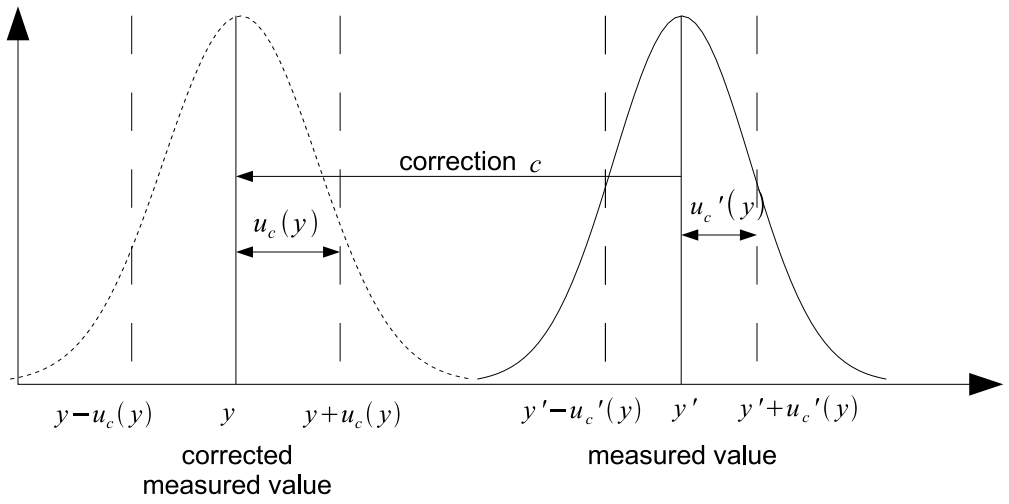


Figure 6: Correcting a measurement

According to the GUM, the model should include corrections compensating systematic effects as input parameters. In order to preserve the linearity of the model the corrections should be included as factors or offsets. In Figure 6 we show a correction of a measured quantity. The measured value y' is shifted by a known systematic effect c. Therefore it needs to be corrected by $-c$. The correction c as well as the other input parameters of the model are input arguments of Equation 2, and the correction c may also be uncertain. Suppose a measurement model f' indirectly obtains y' from a variety of measured input quantities $(x_1, x_2, \ldots, x_n)$. However y' includes a systematic offset c. Equation 3 is used to estimate the true measurement result. This equation should be used to evaluate the combined standard uncertainty according to Equation 2.

$$\begin{aligned} y' &= f'(x_1, x_2, \ldots, x_n) \\ y &= f(x_1, x_2, \ldots, x_n, c) = f'(x_1, x_2, \ldots, x_n) - c \end{aligned} \tag{3}$$

The uncertainty of c is expressed as $u(c)$. The GUM assumes that the majority of applications return an approximately normally distributed output quantity because the central limit theorem (CLT) is met (see Appendix B.3). According to the GUM, the measurement result should be reported as $[\hat{y} \pm u_c(y)]$. The true value is supposed to lie in the interval $[\hat{y} - u_c(y), \hat{y} + u_c(y)]$ of one standard deviation. This representation has a level of confidence of approximately 68%.

However, in some cases it is desirable to state the confidence interval with another level of confidence (e.g. 95% or 99.5%). In these cases the confidence interval is expressed as $[\bar{y} \pm k \cdot u_c(y)]$. The factor k is a so-called coverage factor.

The evaluation of k requires knowledge of u_c that is also only approximated. The GUM proposes to use the Welch-Satterthwaite (W-S) formula to evaluate k based on the degrees of freedom (DOF) ν_i of the estimated input quantities $\hat{x}_i$. The parameter k is then obtained from the quantile function of Students t-distribution (*see* Spiegel and Stephens [21]). The DOF of the input quantities may be interpreted as level of trust. Usually they are equal to the sample size $\nu_i = m$ on which the estimate of the respective input parameter is based on. If a parameter is known

beforehand, such as a systematic effect, the assigned DOF are infinite $\nu_i = \infty$. The W-S formula shown in Equation 4 returns the effective degrees of freedom ν_{eff}.

$$\nu_{\text{eff}} = \frac{u_c^4(y)}{\sum_{i=1}^{n} \frac{\left(\frac{\partial f}{\partial x_i}\right)^4 u^4(x_i)}{\nu_i}} \tag{4}$$

- $u_c(y)$ is the combined standard uncertainty of y that is evaluated using Equation 2.
- $u(x_i)$ is the input uncertainty of the input quantity x_i.
- ν_i is the DOF of the imput quantity x_i, *i.e.* it is evaluated as described above.
- $\frac{\partial f}{\partial x_i}$ expresses the linerized model with respect to x_i.

The effective DOF ν_{eff} and the desired level of confidence p are used arguments of the quantile function of Students t-distribution to obtain the coverage factor k. The final confidence interval is described using $[\hat{y} \pm k u_c(y)]$.

3.3. Software Design

We desire a software design that performs the evaluation of the combined standard uncertainty of y based on $\hat{x}_i$, $u(x_i)$ and systematic effects. Furthermore we require that the software design evaluates the effective DOF automatically. Hall [19] proposes a design pattern encapsulating Equation 2 called GUM–tree. It evaluates the components of uncertainty, maintains global data (i.e. the correlation coefficients) and combines the components of uncertainty. However, his design, presented in [19], does not evaluate the effective degrees of freedom, and is not able to handle physical units by default.

His approach is first described in Section 3.3.1. Thereafter we discuss our design modifying his proposal in Section 3.3.2.

3.3.1. The Hall Proposal

Hall's [19] concept is to break any equation of the form shown in Equation 1 down into a dependency tree of operations. The structure of the tree determines the order of how the operations need to be evaluated to obtain y. To illustrate this procedure consider the following example.

Imagine one has to measure the distance $s(a,t)$ of a uniformly accelerated object after t seconds with a constant acceleration of a. If we assume no initial distance $s(t=0) = 0$ and no initial velocity $v(t=0) = 0$, then Equation 5 gives the distance. Its dependency tree is shown in Figure 7. The plain leaves depict the inputs of the equation, the diagonal-hatched nodes show the operations, and the horizontal-hatched node depicts a constant. The equation can be evaluated using postfix notation. In order to do that, the tree needs to be traversed using the depth-first-search algorithm (see Appendix A). The numbers below the nodes show the order of their evaluation.

$$s(a,t) = \frac{a}{2}t^2 \tag{5}$$

In order to encapsulate the partial derivates, Hall proposes to abstract every subtree as fixed variable replacing each inner node j by a variable as shown in Equation 6.

$$x_j = f_j(\Lambda_j) \tag{6}$$

- x_j is the variable of the current node j.

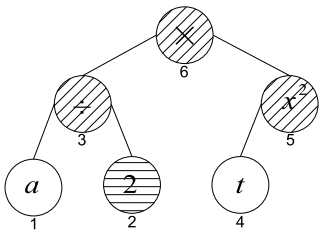


Figure 7: A dependency tree

- Λ_j is the set of parameters that represents the direct child-nodes of the subtrees of the current node.

By applying this scheme to the example described above, we obtain the following equations.

$$\begin{aligned} x_1 &= a \\ x_2 &= 2 \\ x_3 &= f_3(x_1, x_2) = \frac{x_1}{x_2} \\ x_4 &= t \\ x_5 &= f_5(x_4) = x_4^2 \\ x_6 &= f_5(x_3, x_5) = x_3 x_5 \end{aligned} \tag{7}$$

The partial derivates are computed using the chain rule of derivation. Hall defines partial derivates for every inner node using Equation 8.

$$u_j(x) = \sum_{\lambda_k \in \Lambda_j} \frac{\partial f_j}{\partial \lambda_k} u_k(x) \tag{8}$$

- $u_j(x)$ expresses the uncertainty of the current node j with respect to the input parameter x.
- the parameter x is a leaf of the dependency tree.
- λ_k represents a direct child node of the current node.
- $\frac{\partial f_j}{\partial \lambda_k}$ describes the partial derivate of the current node with respect to the child node λ_k.
- $u_k(x)$ expresses the uncertainty of the child node λ_k with respect to the input parameter x. It is evaluated using this scheme recursively.

The partial derivates of the input parameters are computed as shown in Equation 9.

$$u_j(x) = \begin{cases} u(x_j) & x = x_j \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

Note that this representation correctly implements the chain rule for partial derivates. It bears analogy to the partial derivation of a variable with respect to the variable itself and with respect to a constant.

By using the Equations 8 and 9 on our example, we obtain the following partial derivates.

$$u_3(x) = \frac{\partial f_3}{\partial x_1} u_1(x) + \frac{\partial f_3}{\partial x_2} u_2(x)$$

$$= \frac{1}{x_2}u_1(x) - \frac{x_1}{x_2^2}u_2(x) \tag{10}$$

$$u_5(x) = \frac{\partial f_5}{\partial x_4}u_4(x)$$

$$= 2x_4u_4(x) \tag{11}$$

$$u_6(x) = \frac{\partial f_5(x_3,x_5)}{\partial x_3}u_3(x) + \frac{\partial f_5(x_3,x_5)}{\partial x_5}u_5(x)$$

$$= x_5u_3(x) + x_3u_5(x) \tag{12}$$

Hall [19] proposed to generate a class for each elementary function, based on an interface called *IUncertain*. All characteristics such as binary operations, unary operations, input variables, and constants are implemented as subclasses of that interface. Figure 8 shows the classes necessary to evaluate our example.

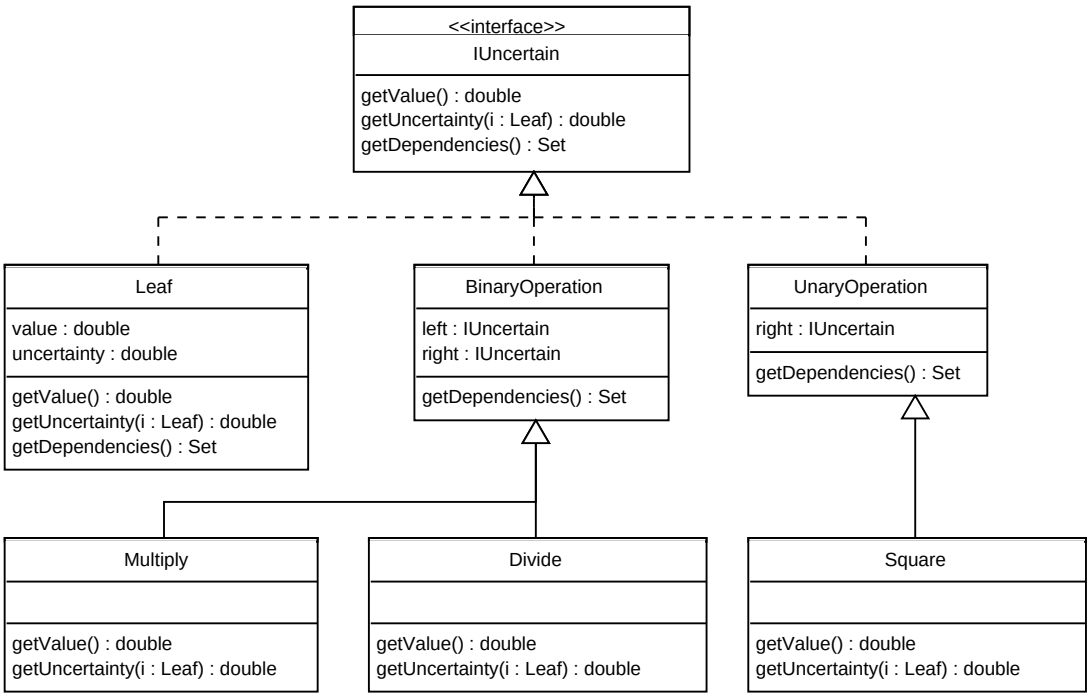


Figure 8: UML diagram of classes necessary to implement the example

- The class `Leaf` is a realization of the interface `IUncertain`. It expresses constants and input variables. Therefore, it has to implement the attributes `value` and `uncertainty`. These are fixed parameters for the measured value and the known uncertainty respectively. The methods `getUncertainty()` and `getValue()` are only accessors to the stored attributes. Since there are no child nodes assigned to a leaf node, the method `getDependencies()` is implemented returning this node only.
- The interface `IUncertain` is the root interface for all nodes. The method `getValue()` is used to return the measured value. The method `getUncertainty()` returns the standard uncertainty. This method implements the partial derivates for the current node, according to Equation 8. Its parameter `Leaf i` is used to allow the partial derivation. The

method `getDependencies()` returns a set of `IUncertain` instances representing the leafs of the current node.
- The classes `BinaryOperation` and `UnaryOperation` provide an abstract interface for binary and unary operations respectively. They contain attributes modelling their subtrees. Thus, the binary operations have left and right subtrees. The unary operations only implement a right subtree.
- The classes `Multiply`, `Divide`, and `Square` are specializations of the binary and unary operations. Each operation has to implement the methods `getValue()` and `getUncertainty()`, which return the calculated value and the uncertainty of their subtree.

The realizations for all operations necessary are stored in a mathematical function library. Besides creating the nodes of the GUM tree, one needs to be able to store global information (such as the correlation matrix) and to invoke the uncertainty calculation. Hall proposes to encapsulate these aspects in a class called `Context`.

Class libraries implementing the pattern described above are *ByGum* [7] and $GUM++$ [22]. From our perspective these are practicable approaches to evaluate the uncertainty as proposed by the GUM [3]. However, the implementations are unable to propagate units along with the uncertainty and the measured value. They do not implement other uncertainty handling approaches (*i.e.* using Bayesian inference, we will describe in Section 5). Furthermore, the standard set of elementary functions is very limited in these class libraries. We desire a class library that has a larger set of elementary functions.

3.3.2. The Reidemeister Formulation

Because of these issues we implemented Hall's [19] proposal again as module in SCUQ, integrating physical quantities having physical units. We also provide an interface for NumPy (*see* Oliphant [6]) that is a very prominent Python library for scientific computing. NumPy has a rich core library of mathematical functions. These functions can be easily integrated into Python classes similar to Pythons operators (see Section 2). Furthermore, our design, like Hall's implementation [7], implements Pythons operators; therefore, SCUQ can be easily integrated into existing Python projects.

We use a similar design as Hall to model the GUM-tree elements shown in Figure 9. Our mathematical function library consists of the classes shown in Figure 10. All uncertain components implement the ufuncs of NumPy shown in Table 6. Furthermore, we implement the mathematical operators shown in Table 2. This allows defining the physical model conveniently.

The class `UncertainInput` models leafs of the GUM-tree. It has the attributes `__uncertainty`, `__value`, and `__dof` expressing the uncertainty, estimated value, and the degrees of freedom. These attributes are available through the methods `get_uncertainty`, `get_value`, and `get_dof`. In analogy to Hall's proposal, the method `depends_on` returns the list of leafs of the current instance. We divided the mathematical functions into two major categories: binary and unary functions. The abstract classes `BinaryOperation` and `UnaryOperation` provide the interface for these function classes.

Our design of the class context is shown in Figure 11. It maintains the correlation coefficients of the input quantities in the hash-table `__corr` and evaluates the effective DOF as well as the combined standard uncertainty. Our class library does not, in contrast to Hall's [7] proposal, require the context to be created before the model. The input quantities are directly created using the default constructor of `UncertainInput`. Furthermore, we implemented static factory meth-

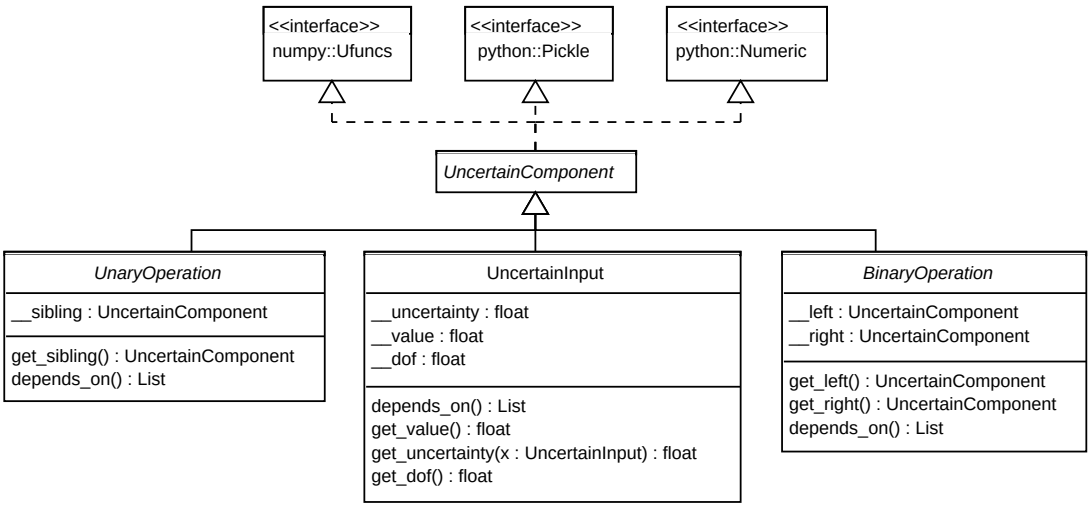


Figure 9: Class diagram of the basic GUM-tree elements

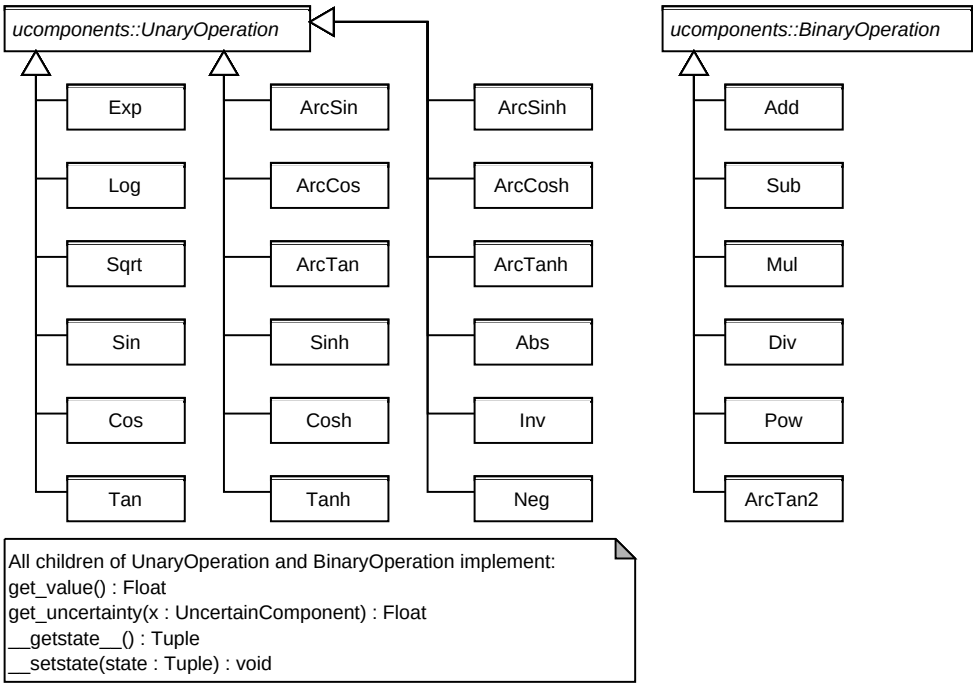


Figure 10: Class diagram of the mathematical functions

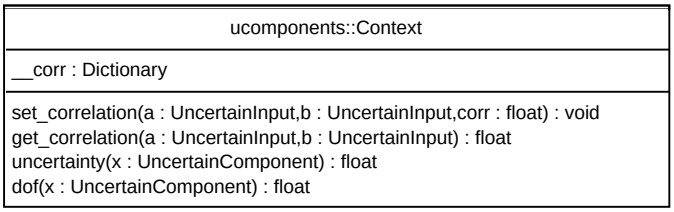


Figure 11: Class diagram of context of the uncertainty evaluation

Method	Description
gaussian(value,sigma,dof)	Creates an instance of `UncertainInput` having $u(x) = \text{sigma}$, $x = \text{value}$, and dof effective degrees of freedom.
uniform(value,halfwidth,dof)	Creates an instance of `UncertainInput` having $u(x) = \frac{\text{halfwidth}}{\sqrt{3}}$, $x = \text{value}$, and dof effective degrees of freedom.
triangular(value,halfwith,dof)	Creates an instance of `UncertainInput` having $u(x) = \frac{\text{halfwidth}}{\sqrt{6}}$, $x = \text{value}$, and dof effective degrees of freedom.
beta(value,p,q,dof)	Creates an instance of `UncertainInput` having $u(x) = \sqrt{\frac{pq}{(p+q+1)(p+q)^2}}$, $x = \text{value}$, and dof effective degrees of freedom.
arcsine(value,dof)	This is an alias for `beta(value, 0.5, 0.5)`.

Table 8: Methods to approximate the uncertainty of various distributions

ods in `UncertainInput` to create leaf nodes for various distributions shown in Table 8. The distributions are described in detail in Appendix B.1.

We demonstrate the use as well as the integration of physical quantities in Section 7. A complete interface description of the classes is available in Appendix C.

3.4. Discussion

In Section 3 we presented a widely accepted standard for the uncertainty propagation, namely the GUM [3]. However, this standard only describes the evaluation of functions that have one scalar output and several scalar input parameters that can be correlated. Furthermore, it proposes the use of the Gaussian error propagation law. This law linearizes the physical model in the environment of the expected output parameter. In general this results only in an approximate solution. The GUM also proposes a technique to define confidence intervals using the Welch-Satterthwaite formula. However, this approach is a proven approximation (*see* Hall and Willink [23]). A technique that propagates the uncertainty more accurately will be discussed in Section 5.

We also presented the GUM tree a software pattern that encapsulates the Gaussian error propagation law. It allows decomposing analytical model functions and evaluates the combined standard uncertainty. Its accuracy is only limited by the destination platform. We have shown that

this pattern can be extended to evaluate the effective DOF of a model as well. Unfortunately, this software pattern has been patented (*see* U.S. Patent 7,130,761 [24]) limiting its application areas.

In Section 4 we present an extension of the GUM tree that can be used to evaluate the uncertainty of complex-valued quantities. We will also show that the results from the evaluation are conforming to the GUM [3]. In Section 7 we provide a variety of examples to demonstrate the use of our design as well as the integration of physical units.

4. Propagation of Uncertainty of Complex-Valued Quantities

We describe the evaluation of uncertainty of complex-valued quantities in this section. The approach described here is an extension of the approach for strictly scalar-valued quantities. Our approach to evaluate the uncertainty of complex-valued quantities extends the proposals of Willink and Hall [25] and Hall [7,18]. We begin by describing our extension of Hall's [7,18] proposals that evaluate of the combined standard uncertainty for complex-valued models in Section 4.1. After that, we present an approach to evaluate confidence regions for complex-valued models in Section 4.2. A conclusion follows in Section 4.3.

4.1. Evaluating the Combined Standard Uncertainty of Complex-Valued Models

In this section we evaluate two approaches to evaluate the uncertainty of complex-valued models. They seem to return different solutions because they have different input parameters. We prove in Appendix B.4 that both approaches return the same result. The standard method we present in Section 4.1.1 expresses the uncertainty of a complex-valued quantity as a special case of an evaluation of an uncertain multivariate quantity. The uncertainty of the influence quantities of a multivariate model are expressed using a covariance matrix. However, the method presented in Section 4.1.2 evaluates the uncertainty using uncertainty of the real- and imaginary part of each influence quantity and their correlation.

4.1.1. The Hall Proposal Based on a Covariance Matrix of the Influence Quantities

The approach described by Hall [18] is based on the adaptation of the Gaussian error-propagation law (*see* Section 3). He extends the Gaussian law of error propagation to bivariate functions and applies this extension to complex-valued quantities. Since the Gaussian error propagation law makes use of partial derivation, the model function must be complex differentiable, and thus meet the conditions of the Cauchy-Riemann equations (see Appendix B.5). Therefore Hall's [18] approach is only applicable to a limited set of functions of all complex functions.

Suppose a measurement process is described by the bivariate function 13 and a countable finite sample set of size n is available that describes the complex-valued inputs of the model.

$$\begin{aligned} \boldsymbol{y} &= \boldsymbol{f}(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_m}) \\ &= f_1(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_m}) + j f_2(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_m}) \\ \boldsymbol{y} &:= y_1 + j y_2 \\ \boldsymbol{x_i} &:= x_{1i} + j x_{2i} \\ y_1, y_2, x_{1i}, x_{2i} &\in \mathbb{R} \\ i &= 1, 2, \ldots, m \end{aligned} \tag{13}$$

- the variables $\boldsymbol{x_i}$ express the complex-valued input parameters.
- the variabe $\boldsymbol{y}$ expresses the complex-valued output parameter.
- Hall models complex functions as bivariate functions. Therefore the complex-valued model function $\boldsymbol{f}$ can be split into a pair of functions f_1 and f_2 that model the real and imaginary part respectively.

Hall's [18] approach is based on the assumption that the complex-valued samples are taken from a bivariate normal distribution (see Appendix B.1.6). In order to propagate the uncertainty

of the input quantities, one has to evaluate the uncertainty of each complex-valued input quantity. Since the samples are generated using a bivariate normal distribution, one has to estimate the mean and the covariance matrix of the distribution of each input quantity. The Formulas 14 show the evaluation of the mean and the covariance matrix based on a countable finite sample of size n set for each input quantity $k = 1,2,\ldots,m$ [18].

$$\begin{aligned} \hat{\boldsymbol{x}}_{\boldsymbol{k}} &= \left[\sum_{i=1}^{n} x_{1ik}, \sum_{i=1}^{n} x_{2ik}\right] \\ \boldsymbol{V}_{\boldsymbol{k}} &= \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{bmatrix} \\ v_{ij} &:= \frac{1}{n(n-1)} \sum_{k=1} n(x_{ijk} - \hat{\boldsymbol{x}}_{ik})(x_{jik} - \hat{\boldsymbol{x}}_{jk}) \quad i = 1,2;\ j = 1,2 \end{aligned} \tag{14}$$

- x_{1ik} expresses the real part of one sample i of the complex-valued input parameter $\boldsymbol{x}_{\boldsymbol{k}}$.
- x_{2ik} expresses the imaginary part of one sample i of the complex-valued input parameter $\boldsymbol{x}_{\boldsymbol{k}}$.
- $\hat{\boldsymbol{x}}_{\boldsymbol{k}}$ is the result of the estimation and expresses the estimated mean value. It is expressed as a two-component vector. Its real- and complex components are expressed using $\hat{\boldsymbol{x}}_{1k}$ and $\hat{\boldsymbol{x}}_{2k}$ respectively.
- $\boldsymbol{V}_{\boldsymbol{k}}$ describes the uncertainty of the complex-valued input parameter $\boldsymbol{x}_{\boldsymbol{k}}$ using a (2×2)-covariance matrix.

Consequently, each input quantity $\boldsymbol{x}_{\boldsymbol{k}}$ is described using an estimated mean value $\hat{\boldsymbol{x}}_{\boldsymbol{k}}$ and an associated covariance matrix $\boldsymbol{V}_{\boldsymbol{k}}$.

Hall describes the Gaussian error-propagation law in matrix-form as shown in the Equations 15. Note that the elements of the vector $\boldsymbol{x}$ are the real parts (x_{1k}) and imaginary parts (x_{2k}) of the input quantities $\boldsymbol{z}_{\boldsymbol{k}}$; $k = 1,2,\ldots,m$.

$$\begin{aligned} \boldsymbol{V}_{\boldsymbol{y}} &= \frac{d\boldsymbol{f}}{d\boldsymbol{x}} \boldsymbol{V}_{\boldsymbol{x}} \left(\frac{d\boldsymbol{f}}{d\boldsymbol{x}}\right)^{T} \\ \frac{d\boldsymbol{f}}{d\boldsymbol{x}} &= \begin{bmatrix} \frac{df_1}{dx_{11}} & \frac{df_1}{dx_{21}} & \frac{df_1}{dx_{12}} & \frac{df_1}{dx_{22}} & \cdots & \frac{df_1}{dx_{1m}} & \frac{df_1}{dx_{2m}} \\ \frac{df_2}{dx_{11}} & \frac{df_2}{dx_{21}} & \frac{df_2}{dx_{12}} & \frac{df_2}{dx_{22}} & \cdots & \frac{df_2}{dx_{1m}} & \frac{df_2}{dx_{2m}} \end{bmatrix} \\ \boldsymbol{x} &= [x_{11}, x_{21}, x_{12}, x_{22}, \ldots, x_{1m}, x_{2m}] \end{aligned} \tag{15}$$

- The matrix $\boldsymbol{V}_{\boldsymbol{x}}$ expresses the input uncertainty. It is the $(2m \times 2m)$-covariance matrix that expresses the input uncertainty of all complex input parameters as well as their covariance. It is composed of $(m \times m)$-sub matrices, denoted as $\boldsymbol{V}_{ij}$, that describe the correlation of the input quantities $\boldsymbol{x}_{\boldsymbol{i}}$ and $\boldsymbol{x}_{\boldsymbol{j}}$ $(i = 1,\ldots,m;\ j = 1,\ldots,m)$. The matrices $\boldsymbol{V}_{ii}$ on the principal diagonal are formed of the matrices $\boldsymbol{V}_{\boldsymbol{k}}$ derived by Formula 14.
- The matrix $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$ is the $(2 \times 2m)$-Jacobian matrix of the model function as shown in Equation 13. The first row contains the partial derivates of the model with respect to the real parts of each input parameter, the second row contains all partial derivates with respect to

the imaginary parts of each input parameter.

- The (2×2)-matrix $\boldsymbol{V_y}$ is the outcome of the computation and expresses the combined standard uncertainty. The principal diagonal contains the squared uncertainty of the real- and imaginary parts of the output parameter. The other elements express the covariance of the real- and the imaginary part.

Since the Cauchy-Riemann equations are met per definition, one can obtain the Jacobian matrix of a complex function using the complex Jacobian matrix of it with respect to its input parameters. Consider the operation $\boldsymbol{f}(\boldsymbol{z_1}, \boldsymbol{z_2}) = \boldsymbol{z_1} \cdot \boldsymbol{z_2}$. Its partial derivates are shown in the Equations 16.

$$\begin{aligned} \frac{\partial f}{d\boldsymbol{z_1}} &= \boldsymbol{z_2} \\ \frac{\partial f}{d\boldsymbol{z_2}} &= \boldsymbol{z_1} \end{aligned} \tag{16}$$

Thus, the complex Jacobian matrix is the matrix $[\boldsymbol{z_2}, \boldsymbol{z_1}]$. Using the Cauchy-Riemann Equations (see Appendix B.5) one obtains the Jacobian matrix shown in Equation 17.

$$\begin{aligned} A_1 &:= Re(z_2) \\ &= a_2 \\ B_1 &:= Im(z_2) \\ &= b_2 \\ A_2 &:= Re(z_1) \\ &= a_1 \\ B_2 &:= Im(z_1) \\ &= b_1 \\ J &= \begin{bmatrix} A_1 & -B_1 & A_2 & -B_2 \\ B_1 & A_1 & B_2 & A_2 \end{bmatrix} \\ &= \begin{bmatrix} a_2 & -b_2 & a_1 & -b_1 \\ b_2 & a_2 & b_1 & a_1 \end{bmatrix} \end{aligned} \tag{17}$$

In order to propagate the uncertainty of the input quantities, an implementation of the algorithm described above must propagate the Jacobian matrices along with the value of the respective input quantities through the model. The model itself can be decomposed into several intermediate steps as described in Section 3. Hall [18] provides a brief example implementation of the decomposition and evaluation of complex uncertainty in C++. The patterns he used are not directly applicable for our purpose. In Hall's [18] example implementation the number of complex-valued input quantities must be known beforehand in order to maintain a global covariance matrix. From our perspective, this limits the convenience of a dynamic model creation. We desire a design that supports modifications to the model without having to reshape the covariance matrix for every modification to the model. From our perspective, it is in practice also rare that uncertain input quantities are always correlated. Thus, a design should also implement this assumption in order to minimize the computation and memory overhead of maintaining a global covariance matrix.

4.1.2. The Hall Proposal Based on Correlation Coefficients of the Influence Quantities

Another approach to propagate uncertainty has also been described briefly by Hall in Appendix C.2 in [7]. Unfortunately, he does not provide any direct reference to [18]. Therefore we validated his approach by showing that his approach proposed in [7] returns the same results as his approach proposed in [18] (*see* Appendix B.4). His approach described in [7] computes the covariance matrix $\boldsymbol{V_y}$ that expresses the combined standard uncertainty more efficiently for a software implementations as the approach described in Section 4.1.1. We show this approach in the Equations 18.

$$\begin{aligned} \boldsymbol{V_y} &= \sum_{i=1}^{m}\sum_{i=j}^{m} \boldsymbol{U_i}(\boldsymbol{y})\boldsymbol{R}_{ij}(\boldsymbol{X})\boldsymbol{U_j}(\boldsymbol{y})^T \\ \boldsymbol{U_j}(\boldsymbol{y}) &= \begin{bmatrix} \frac{\partial f_1}{dx_{1j}} & \frac{\partial f_1}{dx_{2j}} \\ \frac{\partial f_2}{dx_{1j}} & \frac{\partial f_2}{dx_{2j}} \end{bmatrix} \times \begin{bmatrix} u(x_{1j}) & 0 \\ 0 & u(x_{2j}) \end{bmatrix} \\ \boldsymbol{R}_{ij}(\boldsymbol{X}) &= \begin{bmatrix} r(x_{1i},x_{1j}) & r(x_{1i},x_{2j}) \\ r(x_{2i},x_{2j}) & r(x_{2i},x_{2j}) \end{bmatrix} \end{aligned} \tag{18}$$

- The (2×2)-matrix $\boldsymbol{V_y}$ expresses the combined standard uncertainty as it does in the Equations 15.
- The matrix $\boldsymbol{U_j}(\boldsymbol{y})$ models the propagated uncertainty of the influence quantity $\boldsymbol{z_j}$. It is the result of the multiplication of the Jacobian matrix with respect to $\boldsymbol{z_j}$ and the uncertainty of the real part $u(x_{1j})$ and the imaginary part $u(x_{2j})$.
- The matrix $\boldsymbol{R}_{ij}(\boldsymbol{X})$ expresses the correlation between two complex-valued influence quantities $\boldsymbol{z_i}$ and $\boldsymbol{z_j}$.

Using the Formulas 18 instead of the Formulas 15 has several advantages:

1. The computation is carried out using (2×2)-matrices only.
2. The input uncertainty is decomposed into a matrix of the correlation coefficients and the uncertainty of the real and the imaginary part.
3. The correlation of the input quantities can be expressed by a function that returns (2×2)-matrices only.

Point 1 allows hardwiring the matrix operations into software without having to support matrix operations in general. This property reduces the computation effort since dimensional checks of the matrix operands are not necessary. Point 2 allows decomposing the model into its structure and the correlation of the input quantities. Thus, we can store the correlation coefficients separately from the model. Because of point 3, we can implement the supposable sparse correlation matrix as a hash-table and a set of rules.

1. If $i \neq j$ and the input quantities $\boldsymbol{z_i}$ and $\boldsymbol{z_j}$ are uncorrelated, then zero $\mathbf{0}_{2\times 2}$ is returned.
2. If $i = j$ and the real and the imaginary part of the input quantity $\boldsymbol{z_i}$ are uncorrelated, then unity $\boldsymbol{I}_{2\times 2}$ is returned.
3. Otherwise, the (2×2)-matrix of the correlation coefficients is returned. This can be implemented by looking up the matrix of correlation coefficients in a hash-table that has been previously set to the matrix of correlation coefficients of the input quantities $\boldsymbol{z_i}$ and $\boldsymbol{z_j}$.

This approach seems to have a performance drawback since we use a hash-table instead of indexing an array as proposed by Hall in [18]. However, we do not have to fill the covariance matrix initially. We also believe that this performance drawback is ruled out by being able to

create the model dynamically. In contrast to Hall's [18] proposal, we do not have to provide explicit indices for the input quantities in order to obtain their matrix of correlation coefficients. Therefore, we do not have to know the number of complex-valued input quantities beforehand. We also assume that the likelihood of a collision of the hashing-algorithm is very low, because the matrix of correlation coefficients is sparse for the majority of practical applications. Therefore it is from our perspective reasonable to assume that the lookup of the correlation coefficients takes place in constant time (*see* Goodrich and Tamassia [26]).

After having discussed an extension of Hall's [18] approach used for our design, we describe the software structure and the implementation of our design by example.

$$\boldsymbol{y} = \sin(\boldsymbol{z_1} + \boldsymbol{z_2}) \tag{19}$$

As an example, suppose one has the model shown in Equation 19. For the sake of simplicity let's assume that the inputs are uncorrelated. Using the decomposition approach, described in Section 3, one can decompose the equation into the intermediate steps shown in the Equations 20 and 21.

$$\begin{aligned} \boldsymbol{f_1} &= \boldsymbol{z_1} + \boldsymbol{z_2} && (20)\\ \boldsymbol{f_2} &= \sin(\boldsymbol{f_1}) && (21) \end{aligned}$$

In order to propagate the uncertainty of the quantities $\boldsymbol{z_1}$ and $\boldsymbol{z_2}$, one has to obtain the matrices $\boldsymbol{U_j}(\boldsymbol{z_1})$ and $\boldsymbol{U_j}(\boldsymbol{z_2})$ at first. Because of the chain-rule for the derivation of multivariate functions (*see* Wikipedia [27]), the derivation of composite functions is expressed as product of the Jacobian matrices. Let $\boldsymbol{J}(\boldsymbol{f_i})$ denote the Jacobian matrix of the function $\boldsymbol{f_i}$, then one obtains Equation 25 after the intermediate steps shown in the Equations 22, 23, and 24.

$$\begin{aligned} \boldsymbol{J}(\boldsymbol{z_1}) &= \begin{bmatrix} u(x_{11}) & 0 \\ 0 & u(x_{12}) \end{bmatrix} && (22)\\ \boldsymbol{J}(\boldsymbol{z_2}) &= \begin{bmatrix} u(x_{21}) & 0 \\ 0 & u(x_{22}) \end{bmatrix} && (23)\\ \boldsymbol{J}(\boldsymbol{f_1}) &= \boldsymbol{J}(\boldsymbol{z_1}) + \boldsymbol{J}(\boldsymbol{z_2}) && (24)\\ \boldsymbol{J}(\boldsymbol{f_2}) &= \mathrm{M}(\cos(\boldsymbol{f_1})) \times \boldsymbol{J}(\boldsymbol{f_1}) && (25) \end{aligned}$$

This example reveals the requirements of the model building blocks of our design. We need a class that models the input value as well as its input uncertainty. In addition, we need classes that perform the intermediate operations. In Figure 12 we show such a design.

All intermediate operations as well as the input operations are specialized from the abstract class `CUncertainComponent`. Its method `get_value()` provides an interface for the calculation of the value of the component with respect to its inputs. The method `get_a_value()` provides an interface for the conversion of the complex value to a column vector. The method `get_uncertainty(x : UncertainComponent)` provides an interface for calculating the matrix $\boldsymbol{U_j}(\boldsymbol{x})$. The method `depends_on()` provides an interface for returning a list of uncertain input quantities the instance depends on.

There are three specializations of the class `CUncertainComponent`. The classes `CUncertainInput`, `BinaryOperation`, and `UnaryOperation`. The class

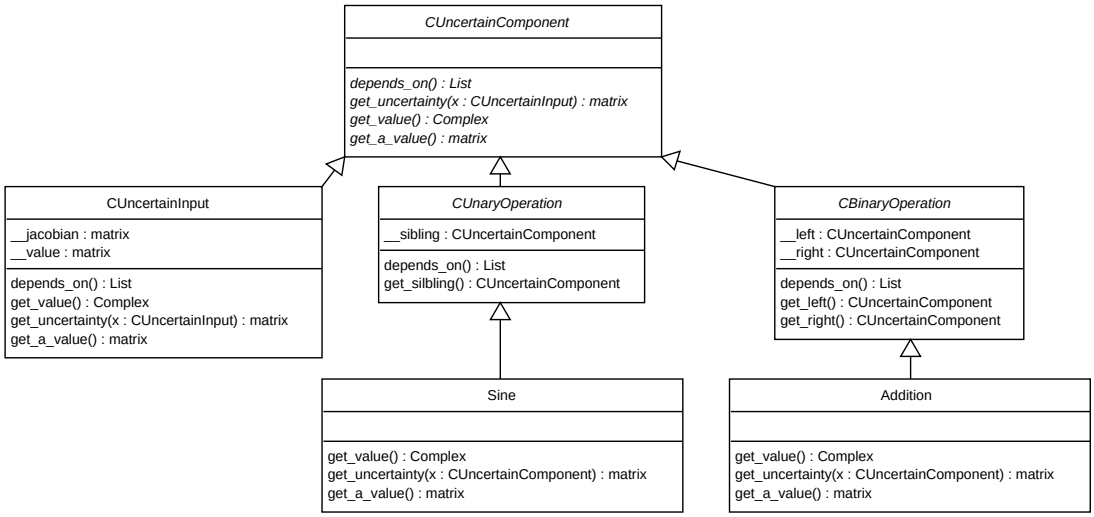


Figure 12: The class hierarchy of our example

`CUncertainInput` expresses the inputs of the model. Its member `__value` stores the input value $z = x_1 + jx_2$ and the member `__uncertainty` stores the input uncertainty as described by Equation 18. The implementation in Python of this class is shown in Listing 4.

```
class CUncertainInput(CUncertainComponent):

  def __init__(self, value, uncertainty):
    self.__value = value
    self.__uncertainty = uncertainty

  def get_value(self):
    return self.__value

  def get_a_value(self):
    return matrix([[self.__value.real],
                   [self.__value.imag]])

  def get_uncertainty(self, x):
    if(x is self):
      return self.__uncertainty
    return zeros((2,2))
```

Listing 4: The Python implementation of the uncertain input quantities

This code sample also shows the implementation of the partial derivation in the method `get_uncertainty()`. If the parameter `x` is identical to this Instance, denoted as `self`, it returns the input uncertainty. Otherwise it returns zero $\boldsymbol{O}_{2\times 2}$. This is an analogous approach for calculating the uncertainty of a variable $\frac{\partial x}{dx}u(x) = u(x)$ and a constant $\frac{\partial x}{dc}u(c) = 0$ for the scalar-valued case.

In order to demonstrate the creation of a Jacobian matrix we present the method `get_uncertainty()` of the class `Sine` in Listing 5. This implementation shows two ma-

jor properties. First, we used the Cauchy-Riemann equations (see Appendix B.5) to derive the Jacobian matrix for the operation $\sin(z)$. Second, we show the application of the chain rule for multivariate functions. The Jacobian matrix of the sine function is expressed in Python by the variable `jac` and the code fragment `self.get_sibling().get_uncertainty(x)` expresses the Jacobian matrix of the sibling. The code shows the implementation of the matrix $U_j(z)$ (shown in the Equations 18).

For the definition of their Jacobian matrices see Appendix B.6.

This scheme is also different from Hall's [19] proposal. He propagated the Jacobian matrices as two-component vector and then applied the transformation, shown in Equation 81, to the result. Therefore his approach is limited to complex functions that fulfil the Cauchy-Riemann equations. In contrast, we can define Jacobian matrices for other complex functions as well.

```
class Sine(UnaryOperation):

  ...

  def get_uncertainty(self, x):
    jac_v = cos(self.get_sibling().get_value())
    jac = matrix([[jac_v.real, -jac_v.imag],
          [jac_v.imag, jac_v.real]])
    return jac * self.get_sibling().get_uncertainty(x)

  ...
```

Listing 5: The Python implementation of the sine-operator

The calculation of the matrix $U_j(z_i)$ for binary functions is done accordingly. We show our implementation of the method `get_uncertainty(self, x)` of the class `Addition` in Listing 6.

We used the same schemes to evaluate the uncertainty of the operations `+,-,*,/,**,~,|` shown Table 2 as well as the NumPy ufuncs, shown in Table 6.

```
class Addition(BinaryOperation):

  ...

  def get_uncertainty(self, x):
    return self.get_left().get_uncertainty(x) \
         + self.get_right().get_uncertainty(x)

  ...
```

Listing 6: The Python implementation of the add-operator

After having described the building blocks of software to calculate $U_j(y)$, we describe the components that allow the computation of the Equations 18. These equations reveal the fundamental requirements for our software design. The evaluation of the combined standard uncertainty as well as the management of the correlation matrix is evaluated using a `Context` class. This is similar to Hall's [18] proposal. He also proposed to use a class `Context` class to manage the covariance matrix of the input quantities. We show our design of the class `Context` in Figure 13.

The methods `set_correlation()` and `get_correlation()` provide an interface to the correlation matrix. We implemented the correlation matrix using a hash-table as described

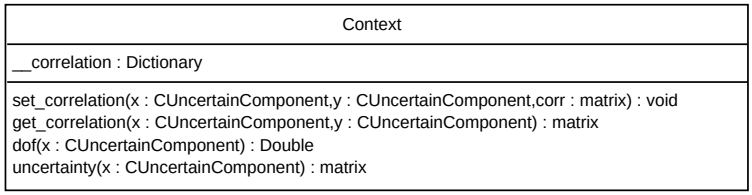


Figure 13: The class Context

earlier in this section. The code of these two methods is shown in Listing 7.

```
1  class Context:

3    ...

5    def set_correlation(self, x, y, corr):
       __correlation[(x,y)] = corr
7
     def get_correlation(self, x, y):
9      if(__correlation.has_key((x,y))):
         return __correlation[(x,y)]
11     else:
         if(x is y):
13         return matrix([[1, 0], [0, 1]])
         else
15         return zeros((2,2))

17   ...
```

Listing 7: The Python implementation of the class Context

The native Python type dictionary implements the hash-table. It is instanced by the variable `__correlation`. The keys of this hash-table are pairs consisting of references to the input quantities (*i.e.* `(x,y)`). Line 6 shows the insertion of a key-value pair into the dictionary. Line 10 shows the retrieval of a value. The code also shows the implementation of the rules that express the correlation matrix as described earlier this section.

```
1  class Context:

3    ...

5    def uncertainty(x):
       inputs = x.depends_on()
7
       sum = zeros((2,2))
9
       for i in inputs:
11       for j in inputs:
           sum += x.get_uncertainty(i) \
13         * self.get_correlation(i,j) \
           * (x.get_uncertainty(j)).T
15
```

```
        return sum
17
    ...
```

Listing 8: The Python implementation of the evaluation of the combined standard uncertainty evaluation for complex-valued models

Method `uncertainty()` performs the combined standard uncertainty evaluation. The code is shown in Listing 8. The summation operators $\sum_{i=1}^{m}\sum_{i=j}^{m}$ are implemented by the two outer loops which iterate over the input quantities of the model expressed by the variable `x`. The correlation matrix $\boldsymbol{R}_{ij}$ is expressed by the call `self.get_correlation((i,j))`. The uncertainty $\boldsymbol{U_i}(\boldsymbol{z_i})$ and the transposed uncertainty $\boldsymbol{U_j}(\boldsymbol{z_j})^T$ are expressed by the calls `x.get_uncertainty(i)` and `(x.get_uncertainty(j)).T`. The variable `u_c` expresses the combined standard uncertainty $\boldsymbol{V_y}$.

4.2. Evaluating Confidence Regions of Uncertain Complex-Valued Models

After having discussed the evaluation of the combined standard uncertainty for complex-valued functions and described our implementation, we shall describe a technique to evaluate the effective degrees of freedom (DOF) of a complex-valued model in this section. The evaluation of the DOF is needed to create confidence intervals/regions for quantities. The approach for scalar-valued models has already been discussed in Section 3.

An approach to calculate the effective degrees of freedom of vector-valued models is discussed by Willink and Hall in [25] and applied to complex-valued models by Hall in Appendix C.3 in [7]. According to Hall [7], the DOF are calculated in two steps. At first the covariance matrices of the input quantities are calculated, shown in Equation 26. After that, the elements of these matrices and the degrees of freedom of the inputs, denoted as ν_i, are used to calculate the effective degrees of freedom, shown in the Equations 27.

$$\boldsymbol{V_i} = \begin{bmatrix} v_{i11} & v_{i12} \\ v_{i21} & v_{i22} \end{bmatrix} = \boldsymbol{U_i}(\boldsymbol{z})\boldsymbol{R}_{ij}(\boldsymbol{X})\left(\boldsymbol{U_j}(\boldsymbol{z})\right)^T \tag{26}$$

$$\begin{aligned} A &= 2\left(\sum_{i=1}^{m} v_{i11}\right)^2 \\ D &= \sum_{i=1}^{m} v_{i11}\sum_{i=1}^{m} v_{i22} + \left(\sum_{i=1}^{m} v_{i12}\right)^2 \\ F &= 2\left(\sum_{i=1}^{m} v_{i22}\right)^2 \\ a &= 2\frac{\left(\sum_{i=1}^{m} v_{i11}\right)^2}{\nu_i} \\ d &= \sum_{i=1}^{m} \frac{\left(v_{i11}v_{i22} + v_{i12}^2\right)}{\nu_i} \\ f &= 2\sum_{i=1}^{m} \frac{v_{i22}^2}{\nu_i} \end{aligned}$$

$$\nu_{\text{eff}} \quad = \quad \frac{A+D+F}{a+d+f} \tag{27}$$

If the conditions of the central limit theorem are met, the effective degrees of freedom can be used in combination with the quantile function of the $F_{2,\nu_{\text{eff}}-2}(\alpha)$-distribution in order to obtain the confidence region. The confidence region is then given by Equation 28. The vector $\boldsymbol{y}$ denotes the value of the model with respect to its inputs and the matrix $\boldsymbol{V_y}$ denotes the combined standard uncertainty. The parameter α expresses the level of confidence.

$$(\boldsymbol{x}-\boldsymbol{y})^T \boldsymbol{V_y} (\boldsymbol{x}-\boldsymbol{y}) \leq \frac{2(n-1)}{(n-2)} F_{2,\nu_{\text{eff}}-2}(\alpha) \tag{28}$$

Since the evaluation of the DOF as well as the evaluation of the confidence region depends on the methods of class `Context`, we also placed the evaluation of DOF into the class `Context` as proposed by Hall [7]. Since there is no analytical way to obtain values from the quantile-function of the $F_{2,\nu_{\text{eff}}-2}(\alpha)$-distribution, we omitted the evaluation of the confidence interval. Our implementation of the method `dof` of the class `Context` is shown in Listing 9. The listing shows that we implemented the two-step procedure as shown in the Equations 26 and 27. The code fragment `i.get_dof()` returns the degrees of freedom assigned to the input quantity `i`.

```
class Context:

  ...

  def dof(x):
    inputs = x.depends_on()

    # Create the covariance-matrices of each input quantitiy
    for i in inputs:
      for j in inputs:
        cov[i] += x.get_uncertainty(i) * self.get_correlation(i,j) * (
            x.get_uncertainty(j)).T

    A = 0.0 ; D = 0.0 ; F = 0.0
    a = 0.0 ; d = 0.0 ; f = 0.0

    # Calculate the coefficients
    for i in inputs:
      A += cov[i][1][1]
      D += cov[i][1][2] ** 2
      F += cov[i][2][2]
      a += cov[i][1][1] / i.get_dof()
      d += (cov[i][1][1] * cov[i][2][2] + cov[i][1][2] ** 2 ) / i.
          get_dof()
      f += cov[i][2][2] / i.get_dof()

    D += A * B
    A = 2.0 * A ** 2.0
    F = 2.0 * F ** 2.0
    a = 2.0 * a
    f = 2.0 * f

    # Return the effective degrees of freedom
```

```
32      return (A + D + F) / (a + d + f)

34  ...
```

Listing 9: The Python implementation of the evaluation of the effective degrees of freedom of complex-valued models

4.3. Conclusion

We discussed the evaluation of complex-valued uncertainties in this section, and we described the evaluation of the combined standard uncertainty that is expressed by a (2×2)-covariance matrix. We also validated Hall's [7] approach to evaluate the combined standard uncertainty. Then we described a technique that can be used to create a confidence region for uncertain complex-valued models. We provided an abstract design as well as implementations in the Python programming language for Hall's proposal using correlation coefficients.

5. Propagation of Uncertainty Using Bayesian Inference

After having described two approaches that make use of the Gaussian error propagation law, in its scalar form (see Section 3) and its bivariate form (see Section 4), we describe another method for propagating uncertainty in this section based on Bayesian inference. This statistical discipline allows incorporating a degree of belief or prior information into the model of a statistical process. Hence, it allows including assumptions about measured values in our case, such as known systematic effects in our model before any data was evaluated. Therefore, statistical effects (Type-A evaluation) and systematic effects (Type-B evaluation) can be included in a common model.

We present and illustrate Weise's and Wöger's [4] method in Section 5.2. Furthermore, we provide an abstract description of a software design that can be used as a roadmap to shape future work projects implementing this method in Section 5.3. We begin introducing Bayesian inference in Section 5.1.

5.1. Fundamentals of Bayesian Statistics

According to Hüllermeier [28], the concept of the Bayesian approach is based on the assumptions shown below:

- probability expresses a degree of belief, no limiting frequency.
- probability statements can be made about parameters of a statistical model, although they are fixed.
- statements of parameter x, such as confidence intervals and expected values, are inferred from a probability distribution $\mathrm{f_x}(x)$. Statistical procedures are assessed by their relative long run frequencies.

Although Bayesian inference uses the same laws of probability calculus, its semantics are different from the classic Frequentists approach shown below [28]:

- probability is described by relative frequencies that are objective properties of the real world.
- parameters are fixed unknown constants; therefore, no useful probability statements can me made about them.

From the viewpoint of the evaluation of the uncertainty in measurements, the following features of Bayesian inference can be concluded and are realized by the method described in [4]:

1. probability statements about the measured data can be made before any data has been seen.
2. artifacts arising from systematic effects can be included in the statistical model.
3. confidence intervals or regions of the measurement result can be directly inferred from the statistical distribution of the measurement model; therefore, they are more or equally accurate compared to those derived with classical approximate methods.

In contrast to point 1, the Frequentist approach does not allow to use information from previous measurements, since the statistical models are assessed by relative long run frequencies. Points 2 and 3 are also novel compared the classical methods described in the Sections 3 and 4 where we used the Welch-Satterthwaite (W-S) formula to evaluate the effective degrees of freedom. These were used to create a coverage factor k describing the confidence interval $[x-ku_c, x+ku_c]$. According to Hall and Willink [23], W-S is only an approximate method. They also show various cases in which W-S returns less accurate confidence intervals than other methods. Because of the approximation using W-S, the Bayesian evaluation of confidence intervals will be more or equally accurate compared to W-S, if the uncertainty arising from systematic effects can be expressed analytically.

Name	Notation	Prior information about the parameter x
Trapeze function	$\mathrm{trapmf}(a,b,c,d,x)$	... defined within $[a,d]$; true value likely within $[b,c]$.
Triangular function	$\mathrm{trimf}(a,b,c,x)$	... defined within $[a,c]$; true value likely to be b.
Rectangular function	$\mathrm{rmf}(a,b,x)$	... defined within $[a,b]$; no information other information given.

Table 9: A selection of fuzzy membership functions

After describing the key features of Bayesian inference, we enumerate the steps of Bayesian inference. According to Hüllermeier [28], inference about a parameter is done in the main steps:

- define a prior distribution $\mathrm{f_p}(\theta)$ that expresses the initial belief about the parameter θ,
- choose a statistical model $\mathrm{f_p}(\theta|x)$ that expresses the belief about the data x given the parameter θ; this function is referred to as *likelihood function*,
- update the degree of belief after observing the data by evaluating the posterior distribution $\mathrm{f_x}(x|\sigma) = c\,\mathrm{f_p}(\theta|x)\,\mathrm{f_p}(\theta)$,
- confidence intervals, the expected value, and the variance can be inferred from the posterior distribution.

In the following chapters we will use the term *prior information* instead of *degree of belief* in order not to imply subjectivity.

5.2. Propagation of Uncertainty

In this section we present the propagation of uncertainty using Bayesian inference, described by Weise and Wöger [4]. At first we describe in Section 5.2.1 the evaluation of the prior distribution based on prior information about the parameters. After that we describe in Sections 5.2.2 and 5.2.3 how the observed data is used to generate the posterior distribution. Finally we show in Section 5.2.4 how to infer the combined standard uncertainty, the expected value and confidence intervals from it.

5.2.1. Prior Information

According to Weise and Wöger [4], the available prior information is the *model prior* and the data prior. The model prior is a set of rules describing prior information available about input and output parameters of a model. It also includes the physical model itself, we describe in Section 5.2.2. We infer from Weise's and Wöger's [4] description that these rules must be defined continuously and limited within 0 and 1.

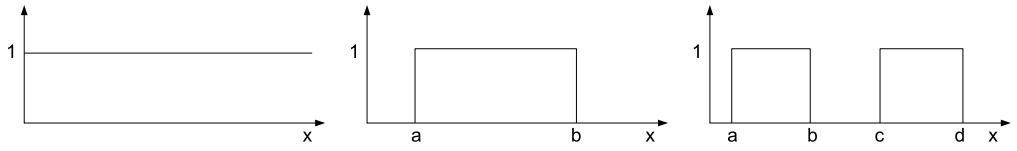
The rules can be interpreted as fuzzy membership functions (*see* Börcsök [29]) and can be combined using fuzzy operators. We provide a list of fuzzy operators in Table 10 and some exemplary membership functions in Table 9. The membership function itself expresses the prior information that is available about a parameter. It may be available as a known systematic effect; for example, the working range of the equipment measuring a parameter. Thus values measured outside of the range are unlikely.

Suppose scalar parameter x is only defined within the interval $[a,b]$. It can be expressed by the rectangular membership function $\mathrm{rmf}(a,b,x)$, shown in Figure 14(b) and implies that all values

Operation	Notation	Definition	Interpretation
Negation	$\neg r$	$1 - r(x)$	The complement of the rule r applies to parameter x.
And	$r_1 \wedge r_2$	$r_1(x)r_2(x)$	The the rules r_1 and r_2 apply to parameter x.
Or	$r_1 \vee r_2$	$r_1(x) + r_2(x) - r_1(x)r_2(x)$	The rules r_1 or r_2 apply to x.

Table 10: A selection of fuzzy operators

within this interval are equally likely. If no prior information exists about a parameter, all values are equally likely. It can therefore be expressed using 1 for all values, shown in Figure 14(a). Suppose we are certain about a parameter being close to the center b of a well-defined interval $[a,c]$. We can express this information using a triangular function $\text{trimf}(a,b,c,x)$. If a parameter x is defined in the interval $[a,b]$ and also in the interval $[c,d]$, then this relation can be expressed by the expression $\text{rmf}(a,b,x) \vee \text{rmf}(c,d,x)$ using the fuzzy $\vee$-operator. The result is shown in Figure 14(c).



(a) The Unity Membership Function (b) The Rectangular Membership Function (c) Using the Or Operator

Figure 14: Several fuzzy membership functions

To express these rules conveniently for all input and output parameters and for other reasons we described in the following sections, Weise and Wöger [4] introduce the vector $\boldsymbol{Z}$ which contains the elements of the vectors of outcomes $\boldsymbol{Y}$ and input parameters $\boldsymbol{X}$. The mapping and its notation are shown in Equation 29. We denote the rules of the model prior as $g(\boldsymbol{Z})$. They contain the prior information about the outcomes and input parameters of the model, except for the physical model itself. We will describe the compilation of the model prior in Section 5.2.2.

$$\boldsymbol{Z} = \{\boldsymbol{Y}, \boldsymbol{X}\} = (y_1, y_2, \ldots, y_m, x_1, x_2, \ldots, y_n) \tag{29}$$

The *data prior* expresses the prior information about the distribution of the input parameters. A multivariate normal distribution is chosen for many technical applications (for a description of the distribution see Appendix B.1.5). Weise and Wöger [4] describe the evaluation of other cases where another distribution was chosen. We limit our description to normally distributed input data.

Because of being able to model systematic effects as fuzzy membership functions, we do not necessarily need to make use of approximate methods, such as W-S (see Section 3) combining statistical effects and systematic effects. If the prior information of a systematic effect can be expressed using Fuzzy rules, the results will be more or equally accurate to the results obtained using W-S and the methods described in Sections 3 and 4. Since W-S is a proven approximationm,

Hall and Willink [23] present various examples where W-S returns less accurate results than other methods, such as Monte-Carlo.

5.2.2. The Likelihood Function Representing the Model Prior

Weise and Wöger [4] refer to the likelihood function as the *model prior*. We already described in Section 5.2.1 that the model prior is a set of rules that describe the prior information about the input and output parameters and the physical model itself.

$$\boldsymbol{y} = f(\boldsymbol{x}) \tag{30}$$

$$\mathbf{0} = \mathrm{M}(\boldsymbol{y}, \boldsymbol{x}) = \mathrm{M}(\boldsymbol{z}) = \boldsymbol{y} - f(\boldsymbol{x}) \tag{31}$$

Suppose our physical model is described by Function 30. Weise and Wöger [4] transform such functions into a form that fits $\mathbf{0}$, shown in Equation 31. They combine the two arguments using Transformation 29 we discussed in Section 5.2.1.

5.2.3. The Posterior Distribution

To infer about the input and output parameters of the model, we need a posterior distribution that expresses our information about the measured values and the uncertainty of them. In this section we describe the evaluation of the measured input quantities and the construction of the posterior distribution.

As we described in Section 5.2.1, we assume that the input data is normally distributed and thus can be expressed by a multivariate normal distribution with the parameters $\mathrm{E}(\boldsymbol{x})$, the vector of expected values of the influence quantities, and the covariance matrix $\mathrm{V}(\boldsymbol{x})$ of the vector of influence quantities $\boldsymbol{x}$ needed to be evaluated. Formula 32 estimates the vector of expected values and Formula 33 estimates the covariance matrix, based on given measurements $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$. These properties can also be a result of a previous evaluation using this method of uncertainty propagation. The data prior would be the obtained posterior distribution and *not necessarily* the multivariate normal distribution. Thus, this method can be applied recursively.

$$\hat{\boldsymbol{x}} = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{x_i} \tag{32}$$

$$\boldsymbol{S} = \frac{1}{n-1} \sum_{i=1}^{n} (\boldsymbol{x_i} - \hat{\boldsymbol{x}}) (\boldsymbol{x_i} - \hat{\boldsymbol{x}})^T \tag{33}$$

$$\mathrm{f_a}(\boldsymbol{z}) = \boldsymbol{C}\, \mathrm{normpdf}(\boldsymbol{x}, \hat{\boldsymbol{x}}, \boldsymbol{S})\, \delta(\mathrm{M}(\boldsymbol{z}))\, g(\boldsymbol{z}) \tag{34}$$

$$\boldsymbol{C} = \int_{-\infty}^{\infty} \mathrm{normpdf}(\boldsymbol{x}, \hat{\boldsymbol{x}}, \boldsymbol{S})\, \delta(\mathrm{M}(\boldsymbol{z}))\, g(\boldsymbol{z}) \partial \boldsymbol{z} \tag{35}$$

The posterior distribution is given by Equation 34. The function $\mathrm{normpdf}(\boldsymbol{x}, \hat{\boldsymbol{x}}, \boldsymbol{S})$ denotes the multivariate normal distribution using the parameters $\boldsymbol{\Sigma} = \boldsymbol{S}$ and $\boldsymbol{\mu} = \hat{\boldsymbol{x}}$. Constant $\boldsymbol{C}$ is a normalizing constant ensuring that the conditions of a probability density functions are met. It can be evaluated using Equation 35.

5.2.4. Reporting the Uncertainty

The expected value $\mathrm{E}(\boldsymbol{z})$ and the covariance matrix $\mathrm{V}(\boldsymbol{z})$ are inferred from $\mathrm{f_a}(\boldsymbol{z})$ using the Equations 36 and 37. Since we included prior information in our model, we also have to report the uncertainty of the input arguments after data has been observed. Therefore, the combined standard uncertainty is expressed by $\boldsymbol{U_z} = \mathrm{V}(\boldsymbol{z})$. It describes the uncertainty of the outcomes, the input parameters and their correlation based on the physical model, the observed data, and prior information (i.e. artifacts arising from systematic effects). The same applies to the expected value.

$$\mathrm{E}(\boldsymbol{z}) = \int \mathrm{f_a}(\boldsymbol{z}) \boldsymbol{z} \partial \boldsymbol{z} \tag{36}$$

$$\boldsymbol{U_z} = \int (\boldsymbol{z} - \mathrm{E}(\boldsymbol{z}))\,(\boldsymbol{z} - \mathrm{E}(\boldsymbol{z}))^T\, \partial \boldsymbol{z} \tag{37}$$

According to Weise and Wöger [4], the posterior distribution $\mathrm{f_a}(\boldsymbol{z})$ should also be included in the report of uncertainty. They suggest the following reasons for consideration:

- the distribution may not be convex.
- the physical model may not be linear within the region of $\mathrm{E}(\boldsymbol{z})$; in this case it may be helpful to use the mode instead of the expected value of $\mathrm{f_a}(\boldsymbol{z})$.

Another reason for including $\mathrm{f_a}(\boldsymbol{z})$ is enabling recursion of the method. The outcomes of this evaluation, namely the expected value $\mathrm{E}(\boldsymbol{z})$, the combined standard uncertainty $\boldsymbol{U_z}$, and the posterior distribution $\mathrm{f_a}(\boldsymbol{z})$, can be used as inputs for another evaluation. In this case $\mathrm{f_a}(\boldsymbol{z})$ expresses the uncertainty of the input quantities, $\mathrm{E}(\boldsymbol{z})$ expresses the expected value of the input quantities, and $\mathrm{f_a}(\boldsymbol{z})$ is the uncertainty of the input arguments.

In order to compare the outcomes of this method to the outcomes of the methods described in the Sections 3 and 4, they have to be marginalized. In Section 3 the combined standard uncertainty as well as the expected value was only evaluated of the output parameter of the model. Therefore, the distribution $\mathrm{f_a}(\boldsymbol{z})$ must be transformed to $\mathrm{f_x}(\boldsymbol{y})$ by integrating out the other parameters, shown in Equation 38.

$$\mathrm{f_x}(\boldsymbol{y}) = \int_{x_1=-\infty}^{\infty} \int_{x_2=-\infty}^{\infty} \dots \int_{x_n=-\infty}^{\infty} \mathrm{f_a}(\boldsymbol{z}) \partial x_1 \partial x_2 \dots \partial x_n \tag{38}$$

The expected value $\mathrm{E}(\boldsymbol{y})$ and the covariance matrix $\mathrm{V}(\boldsymbol{y})$ inferred from $\mathrm{f_x}(\boldsymbol{y})$ have the same dimensions as the outcomes of the methods described in the Sections 3 and 4. Hence, Bayesian inference can be used to propagate the uncertainty of complex-valued quantities as well. In this case the vector $\boldsymbol{z}$ consists of the sequence of real and imaginary parts of the parameters of the model.

5.3. Software Design

Although the method is well described, there are various issues that have to be ruled out using simplifications in order to implement this method in software:

1. At first, we are not aware of a method describing the integrals shown in the Equations 36 and 37 analytically in general; therefore, we need to evaluate them numerically. But a numerical method can only approximate an unbounded integral in general. Therefore a software implementation can only solve problems with a reasonable accuracy that are bounded.

2. All input and output quantities z_i must be described by rules of the model prior. These rules of the must be defined within finite intervals $[a_i, b_i]$.
3. Complex rules can only be defined on single scalar input quantities. All rules are combined using the operator $\wedge$ expressing $g(\boldsymbol{z})$.

The Equation 34 needs to be an integratable function that can be approximated by Monte-Carlo (MC) integration (*see* Appendix B.2).

The simplifications one and two ensure that the integrals shown in the Equations 36 and 37 are always bounded. The third simplification ensures that the numerical integration is applicable. In addition, we limit our design to models having one scalar output parameter only.

Based on these simplifications our software design is divided into the following components:

- classes realizing the model rules and combinations of them,
- a class modelling the input quantities. They contain the model rules regarding the parameter, and the expected value estimated using the observed data,
- several classes to construct the physical model, similar to the GUM-tree explained in Section 3; these classes also evaluate the bounding contour of the integration,
- a class implementing the multivariate normal distribution,
- a class modelling a global context object that maintains the covariance matrix of the input parameters; it also evaluates the posterior distribution to obtain the combined standard uncertainty, and posterior expected value,
- a class implementing a MC integrator; the bounding contour of the MC integrator is defined by the intervals $[a_i, b_i]$ of the model rules of each parameter.

We start by describing the components that realize the model rules. Their class diagram is shown in Figure 15.

Each individual rule has a starting value and an ending value. These values describe the range, in which the rule is nonzero. These properties are available through the methods `get_start` and `get_stop`. Furthermore, each rule assigns a value between $[0, 1]$ to each value of the respective parameter. The method `get_value` performs this mapping. Since these rules are used to limit the bounding contour of the MC integration, they need to be *consistent*. We define a contour being *consistent* if it has a finite volume. Thus, the values returned by the methods `get_start` and `get_stop` must be finite to describe a closed contour. The method `is_consistent` implements this test. The rules describing a single parameter can be constructed by combining several atomic rules. We show some examples in Figure 15. The classes `And`, `Or`, and `Not` implement the respective fuzzy operations (see Table 10) combining model rules. Complex rules can be created by calling the respective methods `and`, `or`, and `not` of an instance of `ParameterRule`.

Atomic building blocks can be implemented directly as sub-classes of `ParameterRule`. Consider the triangular membership function (see Table 10) realized by the class `Triangular`. Each atomic rule has to override the methods `get_value`, `get_start`, and `get_stop`. The triangular membership function is defined by Equation 39. Obviously the parameters a and c are the starting and ending values, and must therefore be returned by `get_start` and `get_stop`. Other atomic rules can be defined in a similar way.

$$\operatorname{trimf}(a,b,c,x) = \begin{cases} \frac{x-a}{b-a} & a \leq x < b \\ \frac{c-x}{c-b} & b \leq x < c \\ 0 & \text{otherwise} \end{cases} \tag{39}$$

After having described the creation of the model rules, we now describe compiling of the physical model. The building blocks are shown in Figure 16. Since we assume that the model can

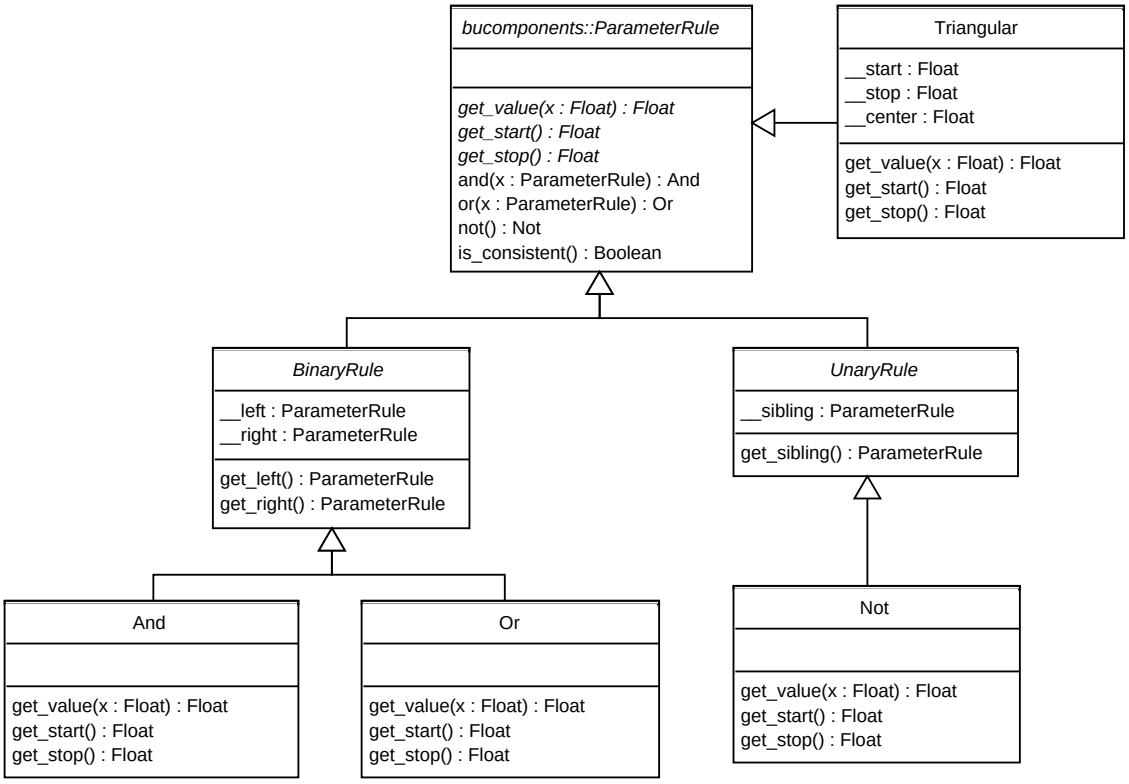


Figure 15: The software design of the rules

be described analytically, we can decompose it into a dependency tree, as we described it in Section 3. The class `UncertainInput` realizes an input parameter of a model. It contains the expected value of the input available from previous evaluations or from the observed data stored in the variable `__expected_value`. Furthermore, the class contains in addition the model rule of the parameter, as described before. Assigning a value to the parameter is required for the MC integrator we describe later this section. The value can be set and retrieved using the methods `set_value` and `get_value`.

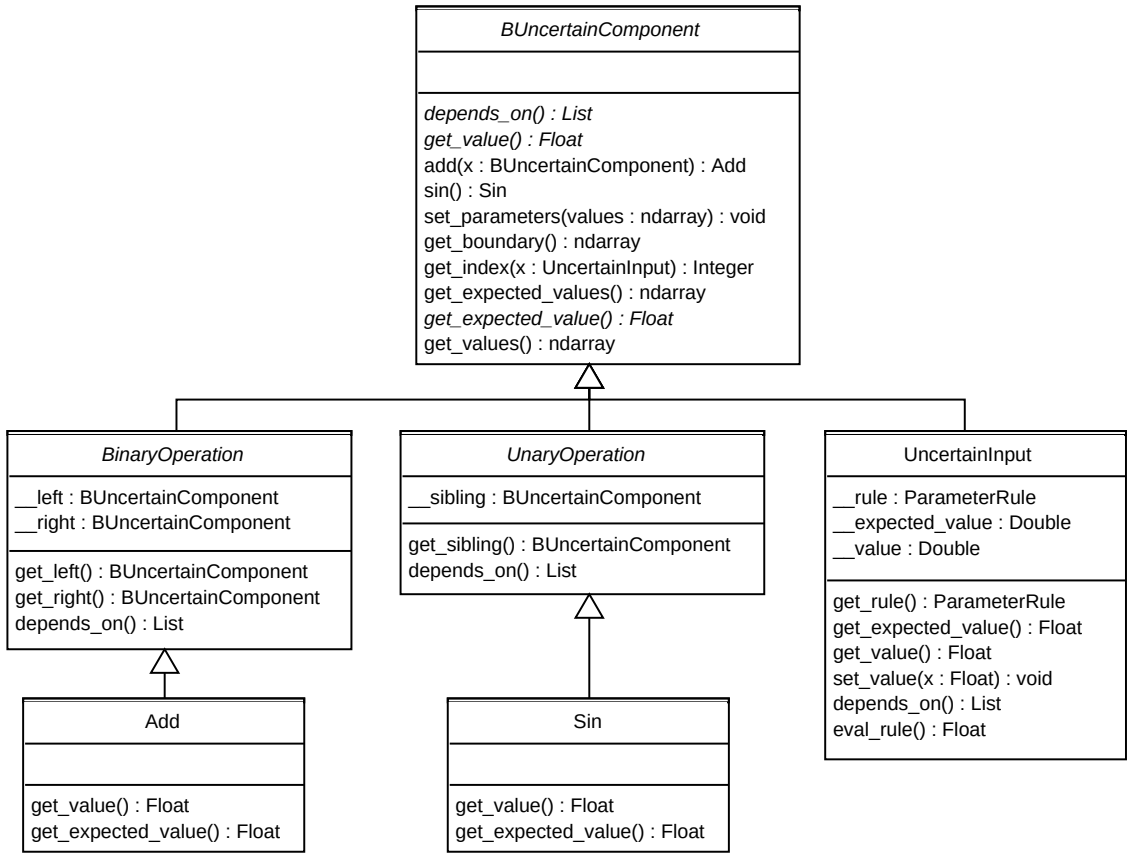


Figure 16: The software components describing the model prior

The input parameters are combined using operators similar to the approach described in Section 3. We included the operators `Add` and `Sin` as examples. All operators must implement the operation `get_value` and `get_expected_value`. These classes propagate the estimated values of the input parameters estimating the expected value to the output parameter of the model.

Moreover, each uncertain component implements the methods `set_parameters`, `get_values`, and `get_expected_values`. These are shortcuts used for the MC integration. The method `set_parameters` sets the parameters of input parameters of the model. The method `get_expected_values` returns the parameters of all inputs. In order to obtain the index of an input quantity to a vector, the method `get_index` can be used. The list of input parameters is available using the method `depends_on`.

After explaining the building blocks of the model, we explain the management of global data such as the covariance of the input parameters and the prior information about the output parameter. Following Hall's [19] proposal, we pack these aspects into the class `Context`, shown in Figure 17. An instance of this class has to be created prior to creating the model.

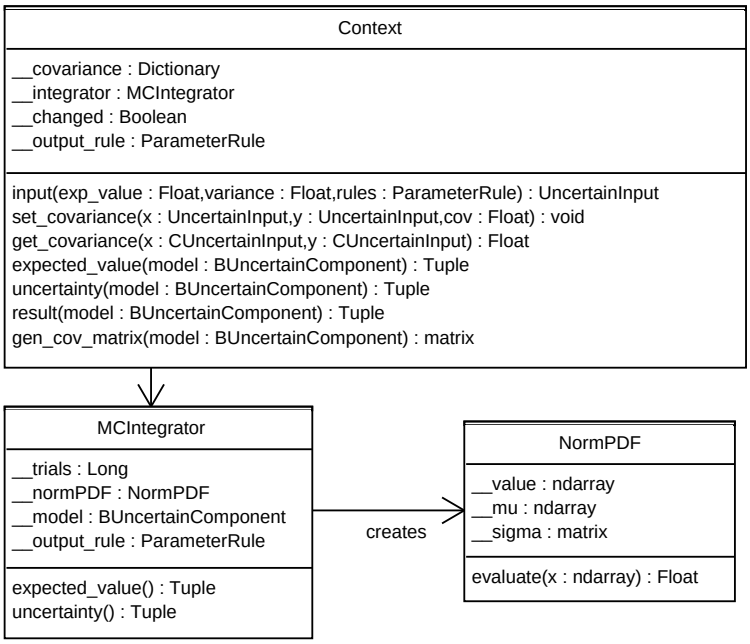


Figure 17: Global information is modelled by the class Context

The input parameters of the model are created using the factory method `input`. Its arguments are the prior expected value of the input parameter, the variance of it (*i.e.* its squared uncertainty) and the rules that express the prior information available. These rules have to be consistent. This property can be checked by calling the method `in_consistent` of the instance of the rule. The variance can be stored in a hash-table that forms the covariance matrix that can be accessed using the methods `set_covariance` and `get_covariance`.

The posterior expected value and the combined standard uncertainty are obtained through the methods `expected_value` and `uncertainty`. The results are reported as tuples. The expected value is reported as a one-dimensional array consisting of the posterior expected value of the posterior distribution and the standard error of the numerical integration. The combined standard uncertainty is reported as quadratic covariance matrix and the standard error of the numerical integration. The abbreviation of both methods is a `result` that returns a tuple containing the posterior expected value and the combined standard uncertainty. These methods invoke the MC integrator. Because MC Integration converges slowly (see Appendix B.2), we assume the integration will consume most computing time while evaluating uncertainty. Therefore, the results should be cached as private members as long as the model is not changed. We introduce the flag `__changed` marking any changes to the model. As long as it is `False` the result of the integration is cached. If it is not set, the integration is performed again. The methods `input` and `set_covariance` unset this flag.

The MC integrator itself uses a multivariate normal distribution to evaluate the integrals shown in the Equations 35, 36, and 37. We propose to perform the integration of all three formulas at once reducing the computation overhead and the standard error, such that the same samples of the parameter space are used as arguments for the three integrals. This procedure is also referred to as *correlated sampling* (*c.f* Banks *et al.* [30]).

5.4. Discussion

Using Bayesian inference has several advantages compared to the classic methods described in the Sections 3 and 4. The Bayesian approach allows incorporating prior information of the parameters into the uncertainty evaluation. This knowledge can be described conveniently by fuzzy membership functions that can be either simple or complex functions created using a Fuzzy algebra. The classic methods can only take into account effects that are approximated by a standard uncertainty, an expected value and a number of degrees of freedom. Thus, the Bayesian method may be equally or more accurate for applications, in which the systematic effects can be described as Fuzzy membership functions. According to Weise and Wöger [4], the outcome of this method are conforming to the GUM [3]. Moreover, we have shown that the results of this method can be adapted to match the results produced by the classic methods described in the Sections 3 and 4.

Furthermore, we analyzed the feasibility of software implementation of the Bayesian approach. We have shown by an abstract UML model that such an implementation is feasible under certain conditions. The biggest issue for a software solution is the evaluation of the integrals shown in the Equations 36 and 37 which are presumably multidimensional with a large number of dimensions. We selected a MC integrator for solving them. Unfortunately this integrator converges slowly (see Appendix B.2), producing a large computational overhead compared to the other methods, described in the Sections 3 and 4.

Our future work will consist of the implementation of the patterns described in the Python programming language [5]. Furthermore, we want to compare different MC integrators for our software design.

Dimension	SI Unit	Symbol
length	meter	m
mass	kilogramm	kg
time	second	s
electric current	ampere	A
thermodynamic temperature	kelvin	K
amount of substance	mole	mol
luminous intensity	candela	cd

Table 11: The SI base-units

6. Units in Measurements

A unit is a quantity having a well-defined property that can be used as factor to express occurring quantities of the same property [31]. We refer to the property as physical dimension. Units based on a fraction of units defined in the same dimension are referred to as *dimensionless units*.

Assume a quantity Y having a known unit $[\mathrm{Y}]$ is measured. It is based on several input quantities $X_1, X_2, \ldots, X_n$ using a model $Y = f(X_1, X_2, \ldots, X_n)$. We assume the units $[\mathrm{X}_i]$ of each input parameter X_i to be known. Thus, the automatic evaluation of the unit $[Y]$ based on the units $[\mathrm{X}_i]$ may be used to verify the plausibility of the measurement model. Furthermore, the GUM [3] requires a statement of the unit of the measurement result.

Therefore, we developed a package modelling unit systems in general and implementing the International System of Units (SI) [32]. We will show in Section 6.3 that our design can be used in place of numerical values in Python, thus, allowing to propagate units along with numeric values or uncertain quantities described in the Sections 3 and 4.

In the next section we provide a brief overview over unit systems by exemplifying the International System of Units (SI). We summarize related work about the implementation of units in soft and hardware in Section 6.2, and describe our design in Section 6.3.

6.1. The International System of Units (SI)

Unit systems consist of a set of fundamental units that are assigned to physical dimensions called base-units. Other units can be derived from these units using operators on them, such as a product of units, multiplication with a constant factor, or adding a constant offset. We show the set of SI base-units and their physical dimension in Table 11. The physical dimensions of the base-units are disjoint. The SI base-units can be realized using an approved list of experiments and reference items (*see* Taylor [32]). The result of a combination of base-units is referred to as *derived* or *coherent unit* and can be represented by another symbol. We show a selection of derived SI units in Table 12.

Unfortunately the statement of a unit is ambiguous. Thus, the physical model cannot be inferred using the unit of a measurement result. Consider the unit surface tension. It can be expressed by $\left[\frac{\mathrm{N}}{\mathrm{m}}\right]$ or by $\left[\frac{\mathrm{kg}}{\mathrm{s}^2}\right]$. Just from the plain statement of the unit one cannot infer if it was obtained by dividing mass by squared time, or by measuring force by length.

This ambiguity is even more challenging for dimensionless units, such as radian [rad] or decibel [dB]. While radian is included as derived unit in the SI describing the dimensionless quantity

Dimension	Name	Symbol	Expression in base-units
plane angle	radian	rad	$\mathrm{m} \cdot \mathrm{m}^{-1}$
frequency	hertz	Hz	s^{-1}
force	newton	N	$\mathrm{m} \cdot \mathrm{kg} \cdot \mathrm{s}^{-2}$

Table 12: A selection of derived SI-units

plane angle, decibel is used to describe various dimensionless quantities on a logarithmic scale. This issue led to the proposal by Mills *et al.* [33] to introduce a system of dimensionless quantities. They argue that the meaning of any quantity depends on the system of equations defining it. Therefore, they propose to introduce a dimensionless quantity one [1]. However, Emerson [34] in contrast to that argues that a unit is just a scale factor and no interpretation of the physical dimension of the value. He concludes that the term plane angle "[...] is generally understood outside a circle of mathematicians and scientists, and is neither dimensionless nor derived [and therefore] it is a base quantity". Furthermore, he proposes to introduce distinct units to model decadic and naperian logarithmic decay "[...] having different applications and [being] expressed in pure numbers in the commonly accepted sense."

Although the argument that units are just scale factors is reasonable from our perspective, we do not agree with his argumentation establishing the units modelling decadic and naperian logarithmic decay. From our perspective the term "commonly accepted sense" is too blurred. There are several different interpretations of the unit decibel in various scientific disciplines, such as electrical engineering (*see* Krauthäuser [35]).

We interpret physical units as scale factors defined in a physical dimension. We assign a physical base dimension to each SI base-unit. These dimensions cannot be renamed, thus the physical dimension is always defined clearly by the canonical form of the product of the base dimensions. In addition, we assume that renaming coherent units allows expressing additional knowledge about a quantity; for example, if one input quantity of a physical model is already measured in Newton then the result of the model maybe affected by a force. We describe a software design in Section 6.3 implementing this assumption.

6.2. Implementing Units into Soft– and Hardware

In this section we discuss related work about the implementation of physical units into soft– and hardware.

The idea of implementing physical units into programming languages has been around for quite a while. The rationale behind this approach is adding constraints to programming languages in order to improve their reliability: "The more constraints a language allows to declare, the more coding errors can be avoided..." (Männer [36]).

Baldwin [37] describes the steps how to modify a Pascal-compiler to check physical units at compile-time. His implementation is based on the assumption that all physical units can be represented as a product of powers of the SI-base-units. Therefore, he proposes to store the powers of the base-units in an array of unsigned chacracters. We refer to this array as *unit array*. This approach is also consistent for dimensionless quantities realized by initializing the unit array to zeroes. Moreover, he proposes to normalize the value of the quantity to the base quantity; for example, 1 [mA] is represented as $1 \cdot 10^{-3}$ [A]. This conversion is done by the preprocessor.

The value is stored as type `REAL` in PASCAL, an implementation of the IEEE-754 floating-point standard (*see* Hennesy and Patterson [38]). However, his design is limited to the operators $+$, $-$, $=$, $\leq$, $\neq$, $\geq$, $<$, $>$, $\times$, and $\div$. He proposes to check the unit array for equality for the operations $+$, $-$, $=$, $\leq$, $\neq$, $\geq$, $<$, and $>$. If the two unit arrays are not equal an exception is raised by the compiler. The multiplication and division of quantities can be realized by adding or subtracting the elements of the unit array. In conclusion, we think this approach is very limited because of the following reasons:

- first, it does not allow an representation of coherent units. Imagine, one would try to express the fuel consumption of a car in liters per 100 [km] using the approach proposed. The base-units are liter ($1[l] = 10^{-3}[m^3]$), for the amount of fuel consumed, and kilometers ($1[\text{km}] = 10^3[\text{m}]$), to express the distance. Because the powers of the normalized quantities are subtracted in this case, the resulting unit is $[\frac{\text{m}^3}{\text{m}}] = [\text{m}^2]$. Although this expression is correct, the resulting unit $[\text{m}^2]$ is, from our perspective, misleading. An implementation of units should also be able to express coherent units.
- second, this approach is limited per definition to a fixed set of operations,
- third, it does not support offset units, such as degrees Celsius ($[^\circ\text{C}] = [K + 273.15]$).

Another similar proposal implementing units in hardware is described in IEEE 1451.2-1997 [39]. IEEE 1451.2-1997 is a standard for a smart transducer interface. *Smart transducers* are transducers that are described by a machine readable interface, have digital control and data channels, and provide triggering, control, and status facilities. A block diagram of a smart transducer interface is shown in Figure 18.

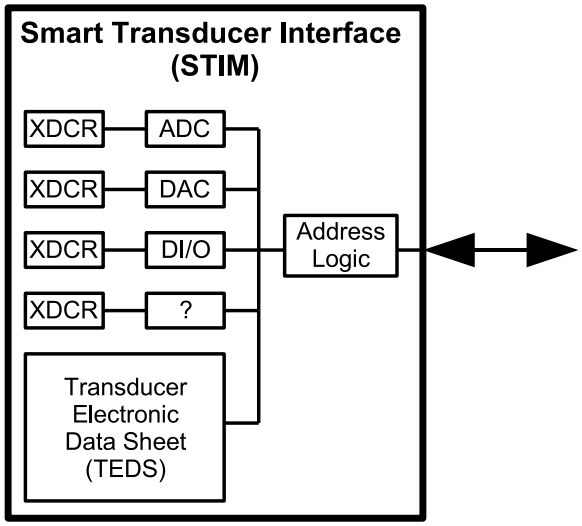


Figure 18: The smart transducer interface module (STIM)

Such transducers have a *Smart Transducer Interface Module* (STIM) that senses or controls a physical quantity. It provides addressing, data transport, status, control, and triggering facilities for one or more channels. Furthermore, the STIM supports calibration and self-test facilities. The functionality of the transducer is described by a *Transducer Electronic Data-Sheet* (TEDS). A TEDS consists of the sections shown in Table 13. The worst-case uncertainty is expressed as combined standard uncertainty and stored as a floating-point value. A lower and an upper bound

Meta TEDS	Channel TEDS	Calibration related TEDS	Extensions
Standard Version	Physical units	Calibration related data	Industry specific extensions
Unique ID	Worst case uncertainty		
Data structure	Range		
Timing	other channel related data		
Channels			

Table 13: The sections of a Transducer Electronic Data-Sheet (TEDS)

value describe the range of the quantity controlled or sensed. The physical units are expressed using a 10 byte vector, shown in Table 14.

Compared to Baldwin's [37] proposal, this implementation also supports an extended description of dimensionless units; for example, it allows expressing the ratio of amperes by ampere $\log_{10}([\mathrm{A}]/[\mathrm{A}])$ on a logarithmic scale. The resulting unit is then described by $(3, 128, 128, 128, 128, 128, \mathbf{130}, 128, 128, 128)$. Baldwin's proposal does not support such an expression. Moreover, the TEDS maintains information about dimensionless quantities. In contrast, Baldwin's proposal uses the canonical form of the product of base-units expressing all dimensionless units the same way. However, both approaches do not support any offset units, such as degrees Celsius ($[^\circ\mathrm{C}] = [\mathrm{K} + 273.15]$).

Another recent proposal integrating units in software is JSR-275 [40] implemented by JScience [41]. JSR-275 proposes a software design incorporating physical quantities in the Java programming language [14]. It is a class library that models physical quantities, physical dimensions, as well as the physical quantities proposed by the SI [32]. It consists of the packages `quantities`, `units`, and `units.converters`. The package `quantities` provides classes modelling the quantities proposed by the SI [32]. We limit our description of the proposal to the packages `units`, and `units.converters`. The package `units` defines the unit types, shown in Figure 19.

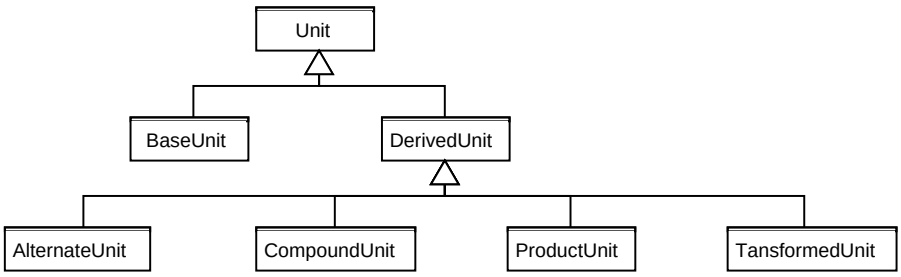


Figure 19: The unit types of JSR-275

All unit types inherit from the abstract type `Unit`. It defines the basic set of unit operations. A unit can be divided and multiplied by other instances of `Unit`. Furthermore, units can be created by applying a constant factor, an exponent, or offset to an existing unit; for example, this allows

Byte	Description
1	Type of unit: 0 – A product of SI base-units, steradians, and radians describe the unit. The powers these units are stored in fields 2 to 10. 1 – The unit is dimensionless and expressed as $\frac{\mathrm{U}}{\mathrm{U}}$, where U is described by the following fields. 2 – The unit is expressed as $\log_{10}(\mathrm{U})$, where the following fields express U. 3 – The unit is expressed as $\log_{10}(\frac{\mathrm{U}}{\mathrm{U}})$, where U is expressed by the following fields. 4 – The physical quantity is digital data and thus all fields shall be set to 128. 5 – The physical quantity is expressed by values on an arbitrary scale. The remaining fields are reserved.
2	$2 \times$ exponent of radians $+ 128$
3	$2 \times$ exponent of steradians $+ 128$
4	$2 \times$ exponent of meters $+ 128$
5	$2 \times$ exponent of kilograms $+ 128$
6	$2 \times$ exponent of seconds $+ 128$
7	$2 \times$ exponent of amperes $+ 128$
8	$2 \times$ exponent of kelvins $+ 128$
9	$2 \times$ exponent of moles $+ 128$
10	$2 \times$ exponent of candelas $+ 128$

Table 14: The TEDS unit definition

expressing degree Celsius based on the SI unit Kelvin ($[^\circ C] = [K + 273.15]$). Moreover, the class `Unit` provides an interface for comparing units using their names, base-units, or their physical dimensions.

The classes `BaseUnit`, `ProductUnit`, and `TransformedUnit` implement a unit system, as described in Section 6.1. The class `BaseUnit` provides an interface for base-units. The units can be defined by assigning a unique symbol an instance of `BaseUnit`. The class `ProductUnit` models units that are created by multiplying instances of `Unit` and storing the result in the canonical form. The class `TransformedUnit` describes units that are created from other units by applying a constant factor, an exponent, or an offset to another unit. These classes inherit from the abstract class `DerivedUnit`, providing an interface for retrieving the parent unit of the current instance.

The class `AlternateUnit` allows renaming derived units, as proposed in by the SI (described in Section 6.1). It assigns a unique symbol to the respective parent unit.

All classes have an interface converting the current instance back to the product of their base-units, or physical dimensions. Therefore, the units can be compared on three levels:

- checking instances of `Unit` for identity, being able to differ between two units that are based on the same base-unit,
- comparing units by the product of their base-units, being able to differ between two units that are created in the same physical dimension,
- being able to compare units using their physical dimension.

Furthermore, JSR-275 [40] allows converting among units of same physical dimension being able to compare different unit systems that share the same physical dimensions; for example, the unit feet can be converted to the unit meters, as shown in Listing 10.

```
import javax.units.*;
import javax.units.converters.*;

class test {
  public static void main(String args[]) {

    /* measure a quantity in meters */
    double value1 = 10; Unit unit1 = SI.METER;

    /* create a converter from meters to feet */
    UnitConverter converter = unit1.getConverterTo(NonSI.FOOT);

    /* convert the value to feet */
    double result = converter.convert(value1);

    System.out.println(""+result+" feet"); // 32.808398950131235 feet
  }
}
```

Listing 10: Example: converting among units that are defined in the same dimension

In line 11, the method `getConverterTo` creates a converter from meter to feet. All unit types implement this method. It converts the units on the different layers described, falling back to a lower level if the conversion on the current level is impossible:

- if both units are equal the method returns identity,
- if the units share the same parent units, it tries to convert the current unit back to its parent

unit and the parent unit forward to the argument,
- finally, the conversion is done using the physical dimensions if all of the above points fail,
- if no conversion is possible, an exception is raised.

The converters are defined in the package `units.converters`.

The proposal JSR-275 [40] implements the SI base-unit as well as the derived SI units. It allows renaming of units to distinguish between quantities of different nature. Furthermore, it allows converting among units of different unit systems that share the same physical dimensions. However, dimensionless units on a logarithmic scale cannot be expressed using their proposal, as IEEE 1451.2 [39] can. From our perspective, JSR-275 is superior compared to the other proposals we reviewed in this section. Because Baldwin's proposal [37] does not allow renaming units, nor does it allow expressing different unit systems. The same drawback applies to IEEE 1451.2. But in addition to Baldwin's proposal, IEEE 1451.2 describes dimensionless units explicitly. Inspired by JSR-275 we describe our implementation of units in the Python programming language [5] in the next section.

6.3. Software Design

After reviewing several implementations of units in hard- and software, we describe in this section our software design implemented in Python [5]. Our design is inspired by JSR-275 [40] that supports compile-time and run-time checks of units of physical quantities in Java [14]. Because Python is a dynamic-typed interpreted language, compile-time checks are from our perspective not feasible. Therefore, we limited our design to run-time verification of physical quantities and units. Furthermore, our design also integrates the approaches we described in the Sections 3 and 4 to propagate the uncertainty in measurements, allowing to assign units to uncertain components that are propagated through a measurement model.

We divided our design into the packages `units`, `operators`, `quantities`, and `si`:

- The package `units` models physical units using the same taxonomy as JSR-275 does (see Section 6.2).
- The package `operators` contains a set of operator classes that are necessary to convert among physical units.
- The package `si` implements the SI base- and derived units.
- The package `quantities` provides the type `Quantity` modelling physical quantities.

The classes modelling physical units are shown in Figure 20.

The class `BaseUnit` provides an interface modelling base-units of a unit system. A unique string identifies each base-unit. Products of units are described by the class `ProductUnit`. Units that are created by applying an offset or factor to an existing unit are expressed using the class `TransformedUnit`. The class `CompoundUnit` provides an interface for compound units as described in the last section. Units can be renamed using the class `AlternateUnit` that assigns a unique symbol to other instances of `Unit`.

In order to define coherent units using other units, we implemented Pythons binary numeric operators `+`, `-`, `*`, `/`, `**` and the unary operator `~`. Furthermore, we also implemented the ufuncs `root`, and `sqrt`. Their definition is shown in Table 15. We use the stereotype $<$ number $>$ to express the native Python types `int`, `long`, `float`, as well as our custom defined type for rational numbers `arithmetic.RationalNumber`. We do not support transformed units using complex factors or offsets. We use the term *parent unit* to express the left operand of binary operations.

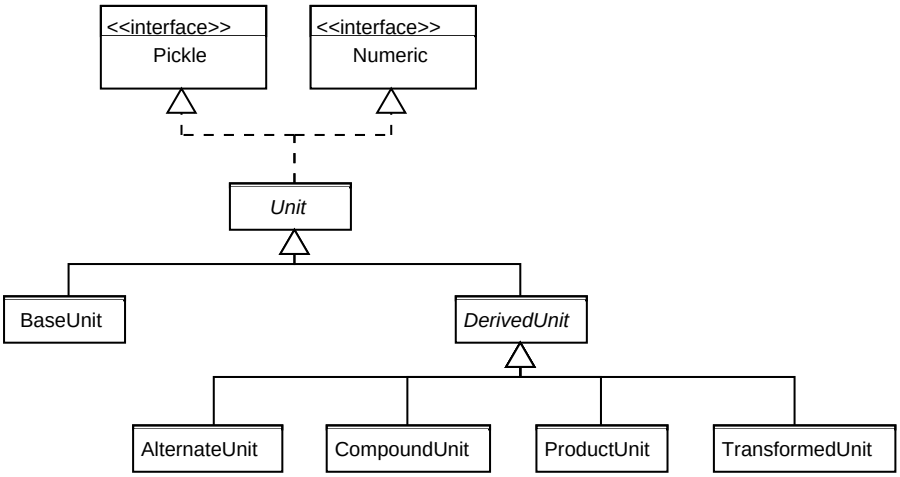


Figure 20: The class hierarchy of the unit types

Operator	Operand Types	Result
+	Unit× < number >	Transformed unit that has a positive offset to its parent unit.
−	Unit× < number >	Transformed unit that has a negative offset to its parent unit.
*	Unit× < number >	Transformed unit factoring its parent unit.
*	Unit × Unit	Product unit of both arguments.
/	Unit× < number >	Transformed unit dividing the parent unit by the argument.
/	Unit × Unit	Product unit representing the fraction of both units.
**	Unit × {long|int}	Product unit representing the n^{th}-power of the parent unit.
~	Unit	Product unit that has inverted exponents; for example $\backsim m^2 = m^{-2}$.
sqrt	Unit	Product unit representing the square root of the argument.
root	Unit × {long|int}	Product unit representing the n^{th}-root of the parent unit.

Table 15: Description of the effect of Python operations on the type unit

The operations are implemented in Python by overriding the respective broadcast method of the operand (see Section 2). An example demonstrating these operators is shown in Listing 11.

[htp]

```
import scuq.units as units
import scuq.si as si

print si.METER                    # ... m
assert(isinstance(si.METER, units.BaseUnit))
```

```
6
SQMETER = si.METER ** 2          # ... define square meter
8 print SQMETER                  # ... m^(2)
assert(isinstance(SQMETER, units.ProductUnit))
10
SQMETER2 = si.METER * si.METER # ... another way
12 print SQMETER2                # ... m^(2)
assert(SQMETER2 == SQMETER)      # ... still square meter
14
CELSIUS = si.KELVIN + 273.15     # ... define celsius on kelvin
16 print CELSIUS                 # ... (K+273.15)
assert(isinstance(CELSIUS, units.TransformedUnit))
```

Listing 11: Example: using Pythons operators to define coherent units

Lines 1 and 2 show import statements of the packages units and si. These statements are necessary to physical units. In this example we import the instance `si.METER` from the module `si` modelling [m]. This unit is already defined as base-unit, as shown in lines 4 and 5. Coherent units can be defined using Pythons operators on the base-unit. Lines 8-9 and 11-13 show different approaches for the definition of the unit square meter using the base-unit meter. Both approaches return equal representations of the unit square meter by creating instances of the class `ProductUnit`. Furthermore, we show in lines 15-17 how transformed units can be created. The unit degrees Celsius can be defined by adding an offset of $+273.15$ to the unit Kelvin.

We also designed the unit comparisons similar to JSR-275 [40]. Units are compared using the following expressions. The expressions `unit1` and `unit2` refer to instances of `Unit`.

1. `unit1 is unit2`
2. `unit1 == unit2`
3. `unit1.is_compatible(unit2)`

These three expressions show the possible layers, on which units are compared. The first expression compares the two units by their object identifiers and returns `True` if they are identical. The second expression compares the units by their definition. The comparison returns `True` if both units are either identical or they have equal representations such that instances of `AlternateUnit` are not equal to instances of `ProductUnit` which share the same physical dimensions.

Suppose a measurement model takes the paint coverage measured in $\left[\frac{\mathrm{m}^2}{\mathrm{l}}\right]$ and the inverse length measured in $\left[\frac{\mathrm{l}}{\mathrm{m}}\right]$ as inputs. Both units are reduced to the same dimension of inverse length. However, the semantics in representation are different [42]. In order not to confuse these two units, we renamed $\left[\frac{\mathrm{m}^2}{\mathrm{l}}\right]$ to [pc] using the class `AlternateUnit` as shown in Listing 12. Although they are still compatible as shown in line 15, they are not equal shown in line 14. In general, we propose to rename special purpose units using the class `AlternateUnit` allowing to preserve the semantics of the representation. Furthermore, this concept may be used to model different dimensionless units.

```
1 import scuq.units as units
import scuq.si as si
3
# Define the unit litre
5 litre = (si.METER ** 3)
litre = litre/1000.0
7
```

```
# Define the unit m^2 / l used for paint coverage
u_paint_coverage = units.AlternateUnit("pc", si.METER ** 2 / litre)

# Define the unit m^{-1}
u_inv_meter = ~si.METER

print (u_paint_coverage == u_inv_meter)                # False
print (u_paint_coverage.is_compatible(u_inv_meter))    # True
```

Listing 12: Example: the difference of dimensional compatibility and equality

The expression shown in Point 3 compares units using their physical dimensions. We already demonstrated this feature in Listing 12. In line 15, the dimension of a unit is obtained using the method `Unit.get_dimension()` based on a physical model. We hardwired the physical model of the SI units into the class `si.SIModel`. This class is derived from the abstract class `units.PhysicalModel`. A custom physical model implementation has to override the method `PhysicalModel.get_dimension(unit)` that returns the physical dimension for each base-unit of the unit system. The module `Units` contains a global variable `default_model` that is interfaced by the method `unit.get_dimension`. The default physical model can be set and accessed using the procedures `units.set_default_model` and `units.get_default_model` respectively. Importing the module `si`, shown in line 2, automatically sets the default physical model to an instance of `si.SIModel`. The implementation of the class `SIModel` is shown in Listing 13.

```
...

class SIModel( units.PhysicalModel ):

    def get_dimension( self, unit ):

        if ( unit == METER ):
            return units.LENGTH
        if ( unit == KILOGRAM ):
            return units.MASS
        if ( unit == SECOND ):
            return units.TIME
        if ( unit == AMPERE ):
            return units.CURRENT
        if ( unit == MOLE ):
            return units.SUBSTANCE
        if ( unit == CANDELA ):
            return units.LUMINOUS_INTENSITY
        if ( unit == KELVIN ):
            return units.TEMPERATURE
        # This should not happen, since we assume that only SI
        # units are used.
        raise qexceptions.UnknownUnitException( "The unit is no SI-
            unit ",
                                                unit )
...
```

Listing 13: The implementation of the class SIModel

Lines 7-20 show the method `PhysicalModel.get_dimension(unit)` returning a predefined dimension for each SI base-unit. We defined the set of base dimensions proposed by the SI [32] in the module `units`. All other physical dimensions are formed of products these base dimensions. The method `Unit.get_dimension` evaluates the dimension of any unit based on its type and the physical model. This method is not overridden by any subclass of `Unit`. That said the given unit types provided by the module `units` are the final set of all unit types.

In addition to the features described above, we also implemented unit conversions. Our design allows comparing of units that share the same physical dimension and physical model. The conversion is implemented in the method `Unit.get_operator_to(unit)` that returns an operator converting values of the given argument to current unit. The method obtains an operator of the current instance back to the product of its base-units and also obtains the same operator for the argument and inverts it. These two operators are then chained to obtain the resulting operator. We illustrate these features in Listing 14 by converting degrees Celsius to Fahrenheit.

```
1 import scuq.units as units
  import scuq.si as si
3
  # Defining Fahrenheit
5 fahrenheit = si.CELSIUS * 9.0 / 5.0 + 32

7 # Fahrenheit to Celsius
  far_cel = si.CELSIUS.get_operator_to(fahrenheit)
9 print far_cel.convert(238)                    # ~100 °C

11 # Celsius to Fahrenheit
  cel_far = fahrenheit.get_operator_to(si.CELSIUS)
13 print cel_far.convert(100)                   # ~238 °F

15 # The Operators can also be inverted using "~"
  print (~far_cel).convert(100)                # ~238 °F
17 print (~cel_far).convert(238)                # ~100 °C

19 # ... and chained using *
  print (cel_far * far_cel).convert(238)       # 238 °F
```

Listing 14: Example: converting degrees Fahrenheit to degrees Celsius

The definition of degrees Fahrenheit in line 5 creates an instance of `TransformedUnit`. This instance internally maintains an instance of `operators.CompoundOperator` that implements the operations dictated by the definition of the unit. The operator that converts values from Fahrenheit to Celsius is created in line 8. It is a combination of the operators shown in Figure 21. The solid arrow shows the order of the definition of the units degrees Fahrenheit and degrees Celsius and the dotted line shows the traversal of the operators to convert from Fahrenheit to Celsius. Note that the operators on the side of Fahrenheit are inverted for this conversion.

In this example the unit Fahrenheit is based on the unit Celsius that is based on the system unit Kelvin. Therefore, the operator created converts Fahrenheit back to Kelvin and Kelvin forward to Celsius. This is the general approach to convert among units of the same dimension. We demonstrated the inversion and chaining of operators in lines 16-20. The Python operator ~ invokes the unit operators method `__invert__` that creates a new instance of an operator that performs the opposite operation. The Python operator $\star$ is used to chain different unit operator instances. The complete set of unit operators is shown in Figure 22.

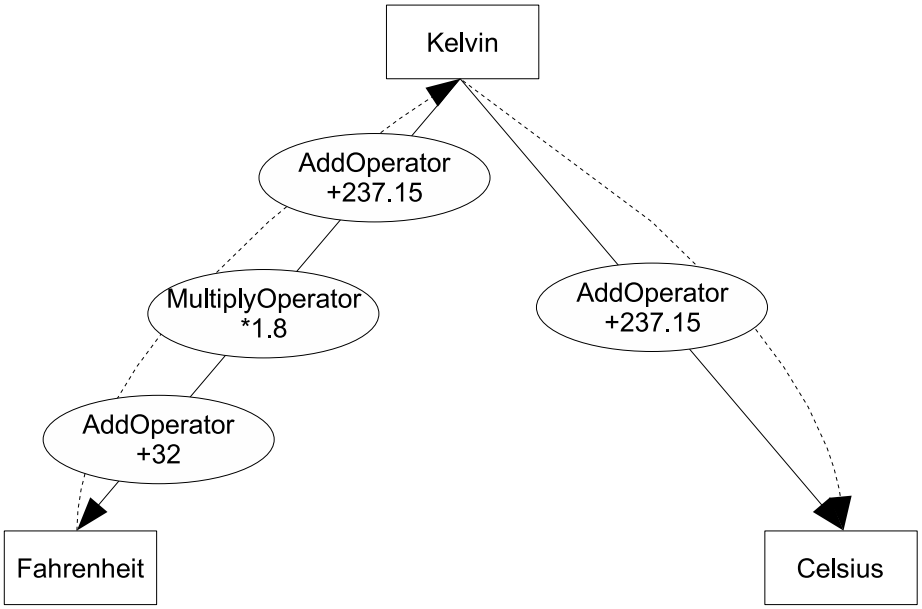


Figure 21: Chaining conversion operators

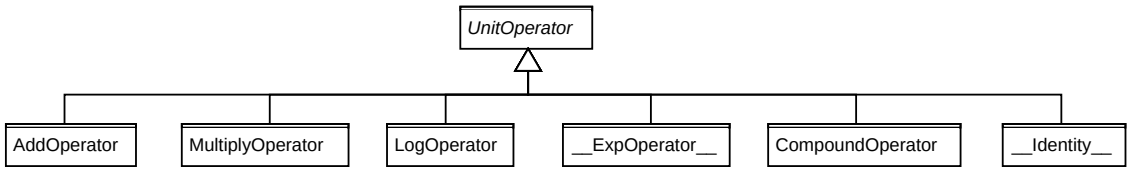


Figure 22: The hierarchy of unit operators

- The class `AddOperator` adds a constant offset to a variable.
- The operator `MultiplyOperator` multiplies variables with a constant offset.
- The class `LogOperator` returns the logarithm of a variable.
- `__ExpOperator__` is a private class that is returned if a LogOperator is inverted.
- The operator `CompoundOperator` implements chains of the operators described.
- The private class `__Identity__` implements identity such that the assigned variable is returned as is. A global instance of this operator is `units.IDENTITY`.

After having described how our class library defines, compares, and converts physical units, we describe our implementation of physical quantities. In contrast to the JSR-275 [40] proposal that defines various different types of physical quantities being checked at compile-time, we implemented a single type `Quantity` that can only be used at run-time. The reason for this obvious drawback is Python being an interpreted language that performs just in time compilations.

We define a physical quantity being a tuple of a value and a *default physical unit*. The term value refers in our definition to numeric values, vectors of numeric values, and uncertain components as described in the Sections 3 and 4. We use a default unit, since the value can be represented in various other units of the same dimension. Based on this definition, we propose the design shown in Figure 23. Furthermore, the type `Quantity` also implements the methods `get_value(unit)`, returning the stored value in the given unit, and the method `get_default_unit` that returns the default unit of the quantity. Moreover, we implemented all numeric operations and NumPy ufuncs described in the Section 2.

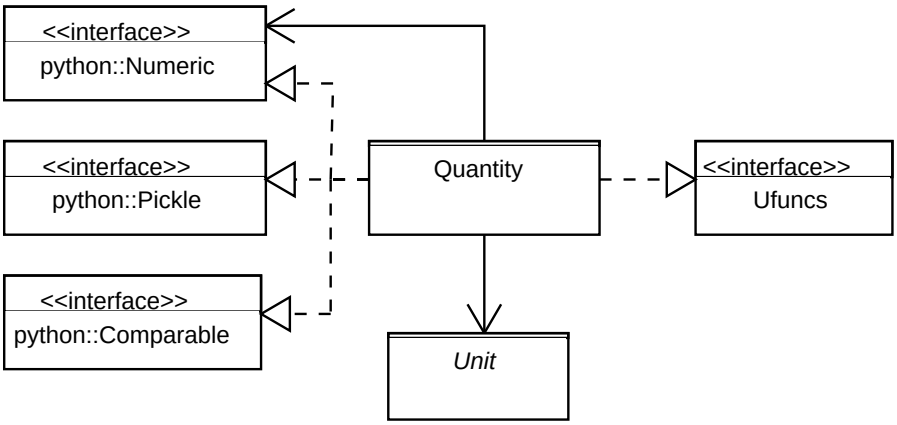


Figure 23: The unit consistency check implemented by the class Quantity

Since, the type `Quantity` implements the Python methods to emulate numeric behavior and the ufuncs of NumPy, it can be easily integrated into existing software that uses native Python types as well as functions of NumPy. The arguments of the binary operations are converted to `Quantity` if possible.

However before any operation is performed on the value, the units of the arguments are compared to ensure consistency. The rules are shown in Figure 24.

The term *comparable* used in Figure 24 denotes the type of the unit comparison. We provide two levels of consistency checking: *weak consistency checking* and *strict consistency checking*.

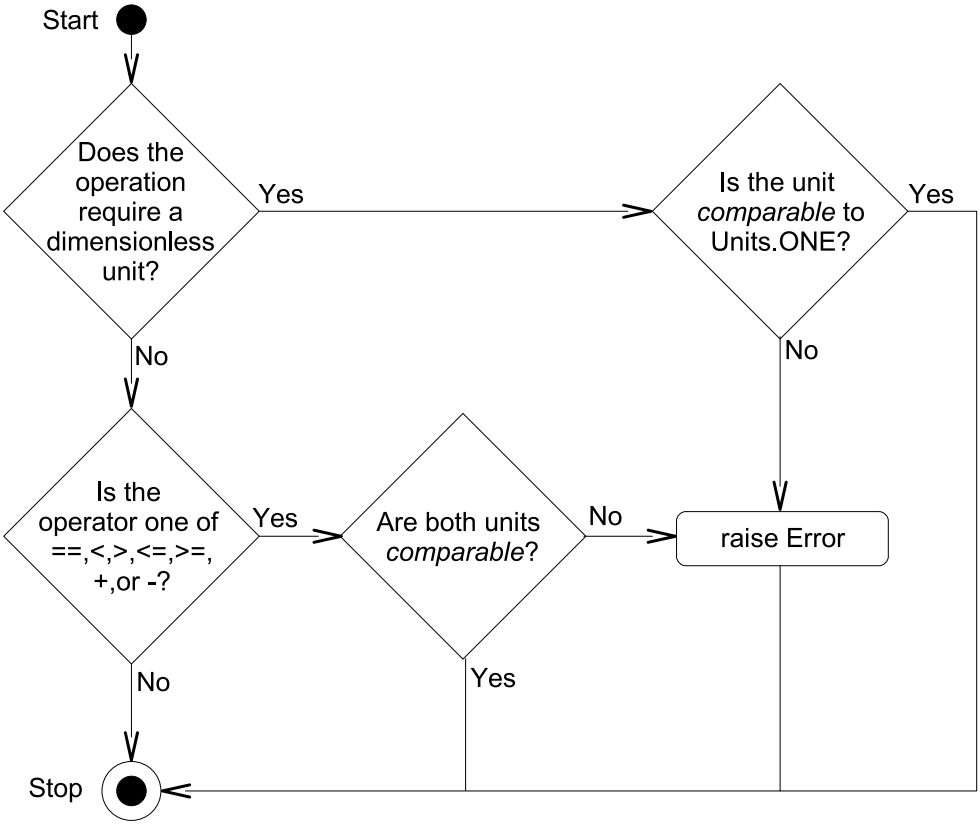


Figure 24: The comparsion of units

Strict consistency checking compares the default units using the operator ==. Thus, the units must mach in type and dimension.

Weak consistency checking compares the units using the method `Unit.is_compatible`. In this case the units are compared using their physical dimensions only. Strict consistency checking is enabled using the procedure `quantities.set_strict(True)` and disabled using `quantities.set_strict(False)` respectively. Checking dimensionless units is also carried out the same way. If strict checking is enabled the unit is checked against `units.ONE` using ==. Otherwise it is only checked for compatibility using `Unit.is_compatible`.

If the units match the rules the operation is performed on the values of the operands of the binary operation. The result is returned as a new instance of `Quantity` that has the default unit of the right hand side operand for binary operations and `Units.ONE` for operations returning a dimensionless quantity. If the unit consistency check fails, an exception is raised. There is however a limitation: If the operands of a binary operation are uncertain quantities the units must be identical. From our perspective, no save conversion is possible because the units describing the uncertainty may have different zero-points.

We provide a complete interface description of the classes described in this Section in Appendix C.

6.4. Discussion

In this section we evaluated the concept of unit systems using the example of SI units. We compared a selection of unit implementations in soft- and hardware, and we presented our own software implementation. From our perspective, unit implementations should provide at least the capabilities to compare units using their physical dimensions. This is also referred to as *dimensional analysis*. Our implementation allows assigning physical units to numeric types in Python. Furthermore it performs automatic consistency checks for Pythons numeric operators as well as a selection of NumPys ufuncs at run-time. Furthermore, we added a layer on top of the dimensional analysis allowing to compare units using their names as proposed by JSR-275 [40]; for example, $\left[\frac{m^2}{l}\right]$ is not confused with $\left[\frac{l}{m}\right]$, if desired. This feature allows assigning a semantic to different units of the same dimension by renaming them. Since our type `Quantity` can store all instances of classes that implement Pythons numeric behavior, we can also store instances of uncertain number types as described in the Sections 3 and 4. In the next section we provide an example how to evaluate the uncertainty of physical quantities using our class library.

Input Quantity	Name of Instance	Distribution of Quantity	Value and Unit of Quantity	Uncertainty	DOF
l_s	`l_s`	Normal	$5 \cdot 10^7$ nm	25 nm	18
d_1	`d_1`	Normal	0 nm	5.8 nm	24
d_2	`d_2`	Normal	0 nm	3.9 nm	5
d_3	`d_3`	Normal	0 nm	6.0 nm	8
α_s	`alpha_s`	Uniform	$11.5 \cdot 10^{-6}$ °C^{-1}	$\pm 2 \cdot 10^{-6}$ °C^{-1}	∞
θ_1	`theta_1`	Normal	0 °C	0.2 °C	∞
θ_2	`theta_2`	Arcsine	0 °C	± 0.5 °C	∞
$\delta\alpha$	`delta_alpha`	Uniform	0 °C^{-1}	$\pm 1 \cdot 10^{-6}$ °C^{-1}	50
$\delta\theta$	`delta_theta`	Uniform	0 °C	± 0.05 °C	2

Table 16: Input quantities used in the end-gauge example, adapted from Hall [7]

7. Examples

After describing different concepts and software designs to propagate the uncertainty in measurements and unit implementations in software, we evaluate a selection of examples in this section. We show how the concepts described the Sections 3, 4 and 6 can be unified to evaluate the uncertainty as well as the units in measurements. The examples we present here are taken from the GUM [1], Appendix H and have already been evaluated by Hall [7]. In Section 7.1 we present the end gauge calibration problem to demonstrate the approach described in Section 3. In Section 7.2, the impedance measurement described in the GUM [1] Appendix H.2, is described. We evaluate it as scalar problem using the approach described in Section 3 and model it as complex-valued problem using the approach described in Section 4.

7.1. End Gauge Calibration Problem

The end gauge calibration problem describes the calibration of a gauge block compared to a standard gauge block. The goal is to determine the uncertainty for the measurement of the length l of the gauge block. The length is evaluated using Equation 40. [7]

$$l = l_s + d - l_s\left(\delta\alpha + \alpha_s + \delta\theta\right) \tag{40}$$

The input parameters of the model are the following:

- the length l_s is the length of the standard gauge block,
- α_s is the thermal expansion coefficient of the standard gauge block,
- $\delta\alpha$ is the difference of the two expansion coefficients of the gauge blocks,
- θ is the temperature offset to 20 °C of the gauge block being calibrated,
- $\delta\theta$ is the temperature difference of the two gauge blocks,
- d is the difference of the lengths of the two gauge blocks being the sum of three contributions d_1, d_2, and d_3.

Table 16 shows the properties of the input quantities and the instances names of the implementation assigned to each quantity.

Our Python code evaluating the problem is shown in Listing 15. We define the units we use in lines 4-7 and create an instance of `Context` that will later be used to evaluate the uncer-

tainty. Then we declare the input parameters, each one in two steps. At first the input quantity is described in terms of distribution and distribution parameters then it is encapsulated in an instance of `Quantity` assigning a unit to it. Consider the definition of the lengths in lines 14-19. At first we define each length as instance of `UncertainInput` using the factory method `UncertainInput.gaussian` and store it in the variable `tmp`. Then this variable is encapsulated in an instance of `Quantity`. We define the other quantities shown in Table 16 accordingly.

```
from scuq import *

# Define the units as transformed units
NANOMETER = si.METER/1e9
CELSIUS   = si.KELVIN+273.15
print "nm := ",NANOMETER
print "C  := ",CELSIUS

c = ucomponents.Context()

tmp = ucomponents.UncertainComponent.gaussian(5e7, 25, 18)
l_s = quantities.Quantity(NANOMETER, tmp)

tmp = ucomponents.UncertainComponent.gaussian(0.0, 5.8, 24)
d_1 = quantities.Quantity(NANOMETER, tmp)
tmp = ucomponents.UncertainComponent.gaussian(0.0, 3.9, 5)
d_2 = quantities.Quantity(NANOMETER, tmp)
tmp = ucomponents.UncertainComponent.gaussian(0.0, 6.7, 8)
d_3 = quantities.Quantity(NANOMETER, tmp)

d = d_1 + d_2 + d_3

# Verify the model
assert(d.get_default_unit() == NANOMETER)

tmp = ucomponents.UncertainComponent.uniform(11.5e-6, 2e-6)
alpha_s     = quantities.Quantity(~CELSIUS, tmp)
tmp = ucomponents.UncertainComponent.uniform(0.0, 1e-6, 50)
delta_alpha = quantities.Quantity(~CELSIUS, tmp)

tmp = ucomponents.UncertainComponent.gaussian(-0.1, 0.2)
theta_1 = quantities.Quantity(CELSIUS, tmp)
tmp = ucomponents.UncertainComponent.arcsine(0.0)
theta_2 = quantities.Quantity(CELSIUS, tmp)
theta = theta_1 + theta_2

# Verify the model
assert(theta.get_default_unit() == CELSIUS)

tmp = ucomponents.UncertainComponent.uniform(0.0, 0.05, 2)
delta_theta = quantities.Quantity(CELSIUS, tmp)

tmp_1 = -l_s * delta_alpha * theta
tmp_2 = l_s * alpha_s * delta_theta

l = l_s + d + tmp_1 + tmp_2
```

```
48 # Verify the model
   assert(l.get_default_unit() == NANOMETER)
50
   print "u(alpha_s)\t\t\t= ",c.uncertainty(alpha_s)
52 print "u(delta_alpha)\t\t\t= ",c.uncertainty(delta_alpha)
   print "u(theta)\t\t\t= ",c.uncertainty(theta)
54 print "u(-l_s * delta_alpha * theta)\t= ",c.uncertainty(tmp_1)
   print "u(l_s * alpha_s * delta_theta)\t= ",c.uncertainty(tmp_2)
56 quantities.set_strict(False) # Enable conversion of units
   print "u(l)\t\t\t\t= ",c.uncertainty(l).get_value(si.METER),si.METER
58 print "dof(l)\t\t\t\t= ",c.dof(l)
```

Listing 15: Example: Python code evaluating the end-gauge problem

After the input quantities have been defined we construct the model in lines 44-46. This step is broken down to intermediate steps to reduce the complexity of the expression. At first we evaluate $-l_s \cdot \delta\alpha \cdot \theta$ and store it in `tmp_1`. Then we construct $l_s \cdot \alpha_s \cdot \delta\theta$ and store it in `tmp_2`. Finally, we evaluate l by adding $l_s + d$ to `tmp_1` and `tmp_2`. In lines 24, 38, and 49 we verify the correctness of the model using the respective units.

We evaluate the combined standard uncertainty, the uncertainties of the parameters, and the effective degrees of freedom in lines 51-58. We obtain the same values as Hall [7] does using his implementation. However, these values differ from the results presented in the GUM [1]. According to Hall, the differences are due to a larger numerical round-off of the results presented in GUM. The output using our implementation is shown in Listing 16. Lines 1 and 2 show the representation of the units [nm] and [°C]. The results presented in lines 3-7 show that the units are maintained correctly. We converted the combined standard uncertainty of the length l from [nm] to [m]. The code for the conversion is shown in lines 56 and 57 of Listing 15. Since strict consistency checking is enabled by default, we need to disable it to convert from [nm] to [m]. Despite both units being realized in the same physical dimension length, [m] is implemented using the class `BaseUnit` and [nm] is created as an instance of `TransformedUnit`. Because both units are not of the same type, the consistency check would have failed if strict checking were enabled (see Section 6.3).

```
   nm :=   (m*1e-09)
2  C  :=   (K+273.15)
   u(alpha_s)                          =  1.15470053838e-06 (K+273.15)^(-1)
4  u(delta_alpha)                      =  5.7735026919e-07 (K+273.15)^(-1)
   u(theta)                            =  0.406201920232 (K+273.15)
6  u(-l_s * delta_alpha * theta)       =  2.88675134595 (m*1e-09)
   u(l_s * alpha_s * delta_theta)      =  16.5988202392 (m*1e-09)
8  u(l)                                =  3.16637674111e-08 m
   dof(l)                              =  16.7521475092
```

Listing 16: Console output of listing 15

This example demonstrated the general procedure to evaluate the uncertainty of uncorrelated scalar quantities. We also demonstrated the use of units in combination with uncertain quantities. In the next section we evaluate another problem showing how a complex-valued problem usually evaluated in scalar terms can be evaluated using the method described in Section 4.

Input Quantity	Name of instance	Value and Unit	Uncertainty
V	v	4.9990 V	0.0032 V
I	i	19.661 mA	0.0095 mA
Φ	phi	1.04446 rad	0.00075 rad

Table 17: Impedance measurement example: estimated input parameters, adapted from Hall [7]

Coefficient	Value
$r(V,I)$	-0.36
$r(V,\Phi)$	$+0.86$
$r(I,\Phi)$	-0.65

Table 18: Impedance measurement example: correlation coefficients of input parameters, adapted from Hall [7]

7.2. Impedance Measurement

Another application area for uncertainty evaluations is impedance measurements in electrical networks. The goal is to evaluate the impedance $\boldsymbol{Z}$, its real R (*a.k.a.* Resistance) and imaginary part X (*a.k.a.* Reactance) based on measurements of the potential difference V, alternating current flowing through a network element I, and their phase Φ. These output parameters are evaluated using the Equations 41.

$$\begin{aligned} R &= \frac{V}{I}\cos(\Phi) \\ X &= \frac{V}{I}\sin(\Phi) \\ \boldsymbol{Z} &= \frac{V}{I} \end{aligned} \tag{41}$$

The estimates of the input parameters are usually taken from the same sample and therefore are correlated. The estimates of the input quantities are shown in Table 17, their correlation coefficients are shown in Table 18.

We evaluate $\boldsymbol{Z}$, R, and X the almost same way as we did in the last Section. The correlation coefficients are maintained by the instance of `Context`. In order to set the correlation of two input quantities `q1` and `q2` , we use the method `set_correlation(q1,q2)`. We show the implementation in Listing 17. Lines 1 and 2 show the import statements. In addition to importing the package `scuq`, we import NumPy [6]. Importing NumPy is necessary to obtain the functions `sin` and `cos`. Furthermore, we define the input parameters in lines 6-11 and the model in lines 14-16. In order to verify the consistency of the model, we ensure the units of the output parameters have the same physical dimension as the SI unit Ohm [Ω]. Finally, we report the uncertainty of the output parameters in lines 28-31.

```
1 from scuq import *
  from numpy import *
```

```
3
c = ucomponents.Context()
5
tmp = ucomponents.UncertainInput.gaussian(4.9990, 0.0032)
7 v   = quantities.Quantity(si.VOLT, tmp)
tmp = ucomponents.UncertainInput.gaussian(19.661e-3, 0.0095e-3)
9 i   = quantities.Quantity(si.AMPERE, tmp)
tmp = ucomponents.UncertainInput.gaussian(1.04446, 0.00075)
11 phi = quantities.Quantity(si.RADIAN, tmp)

13 # Define the model
R   = v / i * cos(phi)
15 X   = v / i * sin(phi)
Z   = v / i
17
# Verify model
19 assert(R.get_default_unit().is_compatible(si.OHM))
assert(X.get_default_unit().is_compatible(si.OHM))
21 assert(Z.get_default_unit().is_compatible(si.OHM))

23 # Correlate input quantities
c.set_correlation(v,i,-0.36)
25 c.set_correlation(v,phi, +0.86)
c.set_correlation(i,phi, -0.65)
27
# Report the uncertainty
29 print "u(R) = ", c.uncertainty(R)
print "u(X) = ", c.uncertainty(X)
31 print "u(Z) = ", c.uncertainty(Z)
```

Listing 17: Python code evaluating the impedance measurement example

We obtained the same results from the uncertainty evaluation as Hall [7] did. However, these results also differ from the results presented by the GUM [1] due to a larger round-off in the results presented in the GUM.

$$\boldsymbol{Z} = V \cdot \exp(j \cdot \Phi) \tag{42}$$

Hall [7] presents another representation for impedance measurements, shown in Equation 42. The impedance, reactance, and resistance are available through the absolute value, the imaginary part, and the real part of $\boldsymbol{Z}$, respectively. Hall's representation is obviously more compact than the model shown in the Equations 41. We evaluate it using the approach described in Section 4. Our implementation is shown in Listing 18. The complex constant j is defined in line 7. The parameters are defined in lines 9-13. The method `gaussian` of the class `Context` takes the estimated value and uncertainty of the real and the imaginary part. Since complex functions are bivariate functions the correlation of the two arguments must be expressed as matrix of correlation coefficients (see Section 4). We use NumPys type `matrix` to express matrices in our implementation.

```
1 from scuq import *
from numpy import *
3
```

```
c = cucomponents.Context()

# Define complex j
_J_ = quantities.Quantity(units.ONE, c.gaussian(0+1j, 0, 0))

tmp = c.gaussian(4.9990, 0.003209, 0.0)
v   = quantities.Quantity(si.VOLT, tmp)
tmp = c.gaussian(19.661e-3, 0.00947e-3, 0.0)
i   = quantities.Quantity(si.AMPERE, tmp)
tmp = c.gaussian(1.04446, 0.0007521, 0.0)
phi = quantities.Quantity(si.RADIAN, tmp)

# Define the model
Z   = v / i * exp( _J_ * phi )

# Verify model
assert(Z.get_default_unit().is_compatible(si.OHM))

# Correlate input quantities
c.set_correlation(v,i,matrix([[-0.36, 0], [0, 0]]))
c.set_correlation(v,phi,matrix([[+0.86, 0], [0, 0]]))
c.set_correlation(i,phi,matrix([[ -0.65, 0], [0, 0]]))

# Report the uncertainty
u_c = c.uncertainty(Z)
print "u(Z) =\n", c.uncertainty(Z)

assert(u_c.get_default_unit().is_compatible(si.OHM**2))

# evaluate u(R) and u(X) explicitly

unit = u_c.get_default_unit()
val = u_c.get_value(unit)
u_r = quantities.Quantity(sqrt(unit),sqrt(val[0,0]))
u_x = quantities.Quantity(sqrt(unit),sqrt(val[1,1]))

print "u(R) = ",u_r
print "u(I) = ",u_x
```

Listing 18: Python code evaluating the impedance measurement example

The results of the complex evaluation of the problem are the complex value of Z with unit $\left[\frac{V}{A}\right]$, and the combined standard uncertainty with unit $\left[\frac{V^2}{A^2}\right]$. The combined standard uncertainties presented in the previous evaluations can be inferred from the covariance matrix as shown in Equations 43. We implemented this conversion in lines 35-38.

$$\begin{aligned} u(\boldsymbol{Z}) &= \begin{bmatrix} u(\Re)^2 & r(\Re,\Im)\cdot u(\Re)^2\cdot u(\Im)^2 \\ r(\Im,\Re)\cdot u(\Re)^2\cdot u(\Im)^2 & u(\Im)^2 \end{bmatrix} \\ u(\Re) &= \sqrt{u(\boldsymbol{Z})}_{11} \\ u(\Im) &= \sqrt{u(\boldsymbol{Z})}_{22} \end{aligned} \tag{43}$$

8. Conclusion

We describe the following in our investigation into approaches propagating the uncertainty in measurements:

- The evaluation of uncertainty of scalar-valued models based on the GUM [3], Hall [19], [20], and Hall and Willink [23].
- The propagation of uncertainty for complex-valued models based on Hall [18], and Willink and Hall [25].
- The propagation of uncertainty using Bayesian Inference based on Weise and Wöger [4] and Hüllermeier [28].
- A software design for object-oriented programming languages propagating the uncertainties modelling the methods enumerated above.
- Selected approaches to integrate units in programming languages and embedded hardware.
- The integration of physical quantities into general-purpose object-oriented programming languages similar to JSR-275 [40] exemplifying the Python programming language.

After having discussed the methods of uncertainty propagation in detail in Sections 3–5, we provide a brief comparison of the approaches for uncertainty propagation in Table 19 that might be used to assist in the choice of a method for uncertainty propagation. We compare the described approaches using the following criteria:

- GUM conformity: Is the approach proposed conforming to the standards of the GUM [1]?
- correlation of input quantities: Does the respective approach evaluate the uncertainty of correlated quantities?
- systematic errors: Is the uncertainty propagation method able to express systematic errors accurately such as a limited measuring range?
- non-linear models: Can the method express non-linear measurement models accurately or does it apply a linear approximation to the model in the region of the estimated value?
- multi-variate problems: Can the method propagate the uncertainty of multivariate problems (*i.e.* multiple output parameters)?
- recursion: Can the method be applied recursively such that the outcomes can be directly used as input parameters of another evaluation using the respective method?
- our software design:
  - true value: Does it estimate the true value?
  - uncertainty: Does it estimate the combined standard uncertainty?
  - confidence interval: Does it estimate confidence intervals?
  - DOF: Does it estimate the effective degrees of freedom?
  - overhead: How high is computation overhead compared to the other methods?

We use the symbols $\surd$ for fully supported, $\diamondsuit$ for partial supported, $\varnothing$ for not supported, and $\diagup$ for not applicable.

The reported research provides a foundation for implementing uncertainty propagation methods in software while using physical units. We hope that the readers enjoy the benefits of a software design of uncertainty propagation methods that implements physical units.

Property	Scalar-Valued Method described in Section 3	Complex-Valued Method described in Section 4	Bayesian Method described in Section 5
Features of our The Method			
GUM Conformity	√	√	√
Correlation of Input Quantities	√	√	√
Systematic Errors	♢	♢	√
Non-linear Models	∅	∅	√
Multivariate Problems	∅	♢	√
Recursion	♢	♢	√
Features of our Software Design			
True Value	√	√	√
Uncertainty	√	√	√
Confidence Interval	∅	∅	∅
DOF	√	√	⁄
Overhead	low	low	high
Correlation of Input Quantities	√	√	√
Systematic Errors	∅	∅	♢
Non-linear Models	∅	∅	∅
Multivariate Problems	∅	♢	∅
Recursion	♢	♢	♢

Table 19: A brief comparision of the methods propagating the uncertainty in measurements

Glossary

analytic An analytic complex valued function is a function that fulfils the Cauchy-Riemann equations, 89

bivariate function A bivariate function f consists of a pair of scalar functions f_1 and f_2., 25

byte compilation Python has a compiler that transforms source code into byte code that is evaluated by the byte-code interpreter. The byte-code is platform independent and it is executed almost as fast as native code., 4

C++ C++ is an object-oriented language available for various platforms., 4, 27

complex differentiable A complex function is complex differentiable if it meets the Cauchy-Riemann equations., 25, 88

DOF Abbreviation for degrees of freedom., 16, 17

dynamic-typed In a dynamic typed programming language the types of all objects are defined at run-time only., 4, 53

GUM Abbreviation for Guide to the Propagation of Uncertainty in Measurements., iii, 15

GUM–tree A software design pattern that models the Gaussian error propagation law., 17

holomorphic A synonyme for analytic, 89

Java Java is an object-oriented language available for various platforms. More information is available at: `http://http://java.sun.com/`, 4, 50, 53

MC The abbreviation for Monte-Carlo, a method of numerical integration., 42, 44–46, 83, 84

NumPy NumPy is a third-party module for scientific computing in the Python programming language., iii, 3, 4, 10, 12–14, 20, 31, 59, 61, 66, 67

Pascal A programing language developed for educational purposes., 48

PDF The abbreviation for *probability density function*., 81–83

Python Python is an object-oriented interpreted scripting language. More information is available at: `http://http://www.python.org/`, iii, 3–5, 8–10, 12–14, 20, 30–32, 35, 46, 47, 53–55, 57, 59, 61, 63, 69

SCUQ The abbreviation for "class library for the automatic evaluation of **S**calar or **C**omplex-valued **U**ncertain **Q**uantities.", vii, 2, 3, 13, 20

SI The abbreviation for the International System of Units., 47

STIM An abbreviation for Smart Transducer Interface Module., 49

TEDS An abbreviation for Transducer Electronic Data-Sheet., 49

ufuncs Universal Broadcasting Functions., 10, 12, 14, 20, 31, 53, 59, 61

W-S The abbreviation for Welch-Satterthwaite formula approximating the effective degrees of freedom., 16, 37

$\cos^{-1}$	The inverse cosine function, 93
$\cosh^{-1}$	The inverse hyperbolic cosine function, 95
$\sin^{-1}$	The inverse sine function, 93
$\sinh^{-1}$	The inverse hyperbolic sine function, 95
$\tan^{-1}$	The inverse tangent function, 94
$\tanh^{-1}$	The inverse hyperbolic tangent function, 96
$\tan_2^{-1}$	The inverse two-argument tangent function, 97
$\mathrm{B}(p,q)$	The beta function, 82
$\bar{z}$	The complex conjugate of z, 90
cos	The cosine function, 92, 93
cosh	The hyperbolic cosine function, 94, 95
$\mathrm{Cov}(\mathbf{X})$	This function returns the covariance matrix of the vector of random variables $\mathbf{X}$, 83
det	This function returns the determinant of a matrix, 83
$\mathrm{E}(X)$	This function returns the expected value of the random variable X, 81–83
exp	The exponential function e^x, 67, 82, 83, 91
f_x	This function denotes a probability density function (PDF), 81–83
$\Im$	This function returns the imaginary part of a complex value or function, 90–98
J	This operator returns the Jacobian matrix of a complex-valued function, 89
J_z	This operator returns the complex Jacobian matrix of a complex-valued function, 89
ln	The logarithmus naturalis (*i.e.* base e), 92, 97
M	This function transforms a vector of complex a complex value into a (2×2)-matrix. If the argument is a complex Jacobian matrix, it is transformed into a scalar Jacobian matrix., 88, 89
Φ	The cummulative distribution function of the standard normal distribution, 84

Pr	This function returns a probability; for example, $\Pr(A)$ returns the probability of the event A., 84
r	This function returns the correlation coefficient of two random variables; for example, $\mathrm{r}(X,Y)$ retuns the correlation coefficient of X and Y., 85–87
$\Re$	This function returns the real part of a complex value or function, 90–98
sin	The sine function, 92
sinh	The hyperbolic sine function, 94
tan	The tangent function, 93
tanh	The hyperbolic tangent function, 95
$\mathrm{u}(X)$	This function returns the standard uncertainty of a random variable X, 81–83
$\mathrm{V}(X)$	This function returns the variance the random variable X, 81–83

References

[1] ISO, "Iso guide 98:1995: Guide to the expression of uncertainty in measurement (gum)," 1995.

[2] DIN, "Din v env 13005: Leitfaden zur angabe der unsicherheit beim messen (in german)," 1999.

[3] B. N. Taylor and C. E. Kuyatt, "Guidelines for evaluating and expressing the uncertainty of nist measurement results," NIST Tecnical Note 1297, Tech. Rep., 1994.

[4] K. Weise and W. Wöger, *Meßunsicherheit und Meßdatenauswertung (in German)*. Weinheim; New York; Chichester; Brisbane; Singapore; Toronto: Wiley-VCH, 1999.

[5] "The python programming language – official website," http://www.python.org/, Nov. 2006.

[6] T. E. Oliphant, *Guide to NumPy*. Travis E. Oliphant, 2006.

[7] B. D. Hall, "'bygum': A python software package for calculating measurement uncertainty," Industrial Research Ltd Report 1305, Tech. Rep., 2005.

[8] E. Schrüfer, *Elektrische Messtechnik (in German)*. München; Wien: Hanser, 2001.

[9] W. Kessel, "Meßunsicherheit, ein wichtiges element der qualitätssicherung (in german)," Physikalisch Technische Bundesanstalt, Tech. Rep., 1999.

[10] J. Oberg, "Why the mars probe went off course," *IEEE Spectr.*, vol. 36, no. 12, pp. 34–39, 1999.

[11] "Metrodata gmbh : Gum workbench," http://www.metrodata.de/, Feb. 2007.

[12] "Ptb jahresbericht abteilung 8 (2001)," http://www.ptb.de/de/publikationen/jahresberichte/jb2001/oe8/nachrichtenausabteilung/abteilung8_12.htm, Feb. 2007.

[13] R. Gupta, *Making Use of Python*. New York: Wiley Publishing, Inc., 2002.

[14] "Java technology," http://java.sun.com/, Nov. 2006.

[15] "Microsoft windows family homepage," http://www.microsoft.com/windows/, Nov. 2006.

[16] "Gnu software foundation," http://www.gnu.org/, Nov. 2006.

[17] "Apple : Mac os," http://www.apple.com/macosx/, Nov. 2006.

[18] B. D. Hall, "Calculating measurement uncertainty for complex-valued quantities," *Measurement science and technology*, no. 14, pp. 368–375, 2003.

[19] ——, "The 'gum tree': A software design pattern for handling measurement uncertainty," Industrial Research Ltd Report 1291, Tech. Rep., 2003.

[20] ——, "Calculating measurement uncertainty using automatic differentiation," *Measurement Science and Technology*, vol. 13, pp. 421–427, 2002.

[21] M. R. Spiegel and L. J. Stephens, *Statistik (in German)*. Bonn: mitp-Verlag, 2003.

[22] B. D. Hall, "'gum++': A tool for calculating measurement uncertainty in c++," Industrial Research Ltd Report 1303, Tech. Rep., 2005.

[23] B. D. Hall and R. Willink, "Does "welch-satterthwaite" make a good uncertainty estimate?" *Metrologia*, no. 38, pp. 9–15, 2001.

[24] B. D. Hall and R. D. Willink, "Uncertainty propagation system and method," *U.S. Patent 7,130,761*, Oct. 2006.

[25] R. Willink and B. D. Hall, "A classical method for uncertainty analysis with multidimensional data," *Metrologia*, no. 39, pp. 361–369, 2002.

[26] M. T. Goodrich and R. Tamassia, *Algorithm Design*. Singapore: John Wiley and Sons Inc., 2003.

[27] "Chain rule," http://en.wikipedia.org/wiki/Chain_rule, Nov. 2006.

[28] E. Hüllermeier, "Lecture notes: Computational statistics," http://wwwiti.cs.uni-magdeburg.de/, Otto-von-Guericke University, Magdeburg, Germany, 2005.

[29] J. Börcsök, *Fuzzy Control: Theorie und Industrieeinsatz (in German)*. Berlin: Verlag Technik, 2000.

[30] J. Banks, J. Carson, and B. L. Nelson, *Discrete-event system simulation*. Prentice-Hall, 1996.

[31] "Units of measurement," http://en.wikipedia.org/wiki/Units_of_measurement, Nov. 2006.

[32] B. N. Taylor, "The international system of units (si)," NIST Special Publication 330, Tech. Rep., 2001.

[33] I. M. Mills, B. N. Taylor, and A. J. Thor, "Definitions of the units radian, neper, bel and decibel," *Metrologia*, vol. 38, pp. 353–361, 2001.

[34] W. H. Emerson, "A reply to "definitions of the units radian, neper, bel and decibel" by i. m. mills et al." *Metrologia*, vol. 39, pp. 105–109, 2002.

[35] H.-G. Krauthäuser, "Lecture notes: Emv messtechnik (in german)," http://www.uni-magdeburg.de/iget/, Otto-von-Guericke University, Magdeburg, Germany, 2005.

[36] R. Männer, "Strong typing and physical units," *SIGPLAN Not.*, vol. 21, no. 3, pp. 11–20, 1986.

[37] G. Baldwin, "Implementation of physical units," *SIGPLAN Not.*, vol. 22, no. 8, pp. 45–50, 1987.

[38] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design – The Hardware / Software Interface*. San Francisco, CA, USA: Morgan Kaufmann, 1997.

[39] *IEEE Standard for Smart Transducer Interface for Sensors and Actuators - Transducer to Microprocessor Communication Protocols and Transducer Electronic Datasheet (TEDS), IEEE 1451.2*, IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1997.

[40] *JSR 275: Units Specification*, http://jcp.org/en/jsr/detail?id=275, Java Community Process, June 2005.

[41] "Jscience – java tools and libraries for the advancement of sciences," http://www.jscience.org/, Nov. 2006.

[42] B. D. Hall, "Software support for physical quantities," pp. 66–71, 2002.

[43] V. K. Balakrishnan, *Graph Theory*. McGraw-Hill, 1997.

[44] R. Sedgewick, *Algorithms in C++*. Addison-Wesley, 1992.

[45] "Dreiecksverteilung (in german)," http://de.wikipedia.org/wiki/Dreiecksverteilung, Nov. 2006.

[46] "Betaverteilung (in german)," http://de.wikipedia.org/wiki/Beta-Verteilung, Nov. 2006.

[47] W. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C – The Art of Scientific Computing*. Cambridge; New York; Port Chester; Melbourne; Sydney: Cambridge University Press, 1992.

[48] S. Lipschutz, *Theory and Problems of Linear Algebra*. McGraw-Hill Inc., 1977.

[49] "The wolfram mathworld site," http://mathworld.wolfram.com/ComplexDifferentiable.html, Nov. 2006.

[50] "The wolfram functions site," http://functions.wolfram.com/, Nov. 2006.

A. The Depth-First-Search (DFS) Algorithm

In this section we describe the Depth-First-Search (DFS) Algorithm. It is used to traverse nodes in an connected graph. A special application for this algorithm is the traversal of trees. In Listing 19 we show how a tree is realized in *C*. Each node contains a value *value* and has *numSubTrees* subtrees that can be reached trough the pointer array *aSubTrees*.

```
struct Node {
  /* value of the node, i.e. a String */
  char * value;
  /* number of subtrees */
  int numSubTrees;
  /* pointers to subtrees */
  struct Node **aSubTrees;
}
```

Listing 19: Implementation of a tree-node in C

The tree can be traversed by the code shown in Listing 20. It traverses each subtree recursively, before the current node. Visiting a node can be implemented by implementing the function `visit(struct *Node n)`. The recursion stops as soon a node is traversed that does not have any subtrees (i.e. *numSubTrees* is equal to 0).

```
void traverse(struct Node *node) {
  /* iterate over all subtrees */
  for(int i=0; i < node->numSubTrees; i++) {
    /* traverse each subtree */
    traverse(node->aSubTrees[i]);
  }

  /* after all subtrees are visited, visit the current node */
  visit(node);
}
```

Listing 20: Implementation of a DFS-traversal in C

The general approach to traverse trees is described by Balakrishnan [43]. Examples for implementations to traverse binary trees and graphs are given by Sedgewick [44].

B. Mathematical Proofs and Formulas

B.1. Selected Statistical Distributions

In this section we provide an overview of selected probability distributions and their properties. All distributions, except the bivariate– and multivariate normal distribution, describe scalar random variables. The symbols $\mathrm{E}(X)$ and $\mathrm{V}(X)$ express the expected value and variance of a random variable X being distributed by the probability density function (PDF) $\mathrm{f_x}(X)$. Their evaluation is shown in the Equations 44. Note that the standard uncertainty $\mathrm{u}(X)$ of a random variable X being distributed by a PDF $\mathrm{f_x}(X)$ is equal to the square root of its variance ($\mathrm{u}(X) = \sqrt{\mathrm{V}(X)}$).

$$\begin{aligned} \mathrm{E}(X) &= \int_{-\infty}^{\infty} x \mathrm{f_x}(X) dX \\ \mathrm{V}(X) &= \mathrm{E}(x - \mathrm{E}(X))^2 \\ &= \mathrm{E}(x^2) - (\mathrm{E}(X))^2 \end{aligned} \tag{44}$$

B.1.1. Uniform Distribution

The Uniform distribution is expressed by the PDF 45 of a random variable X that can take values in the interval (a,b) only (*see* Weise *et al* [4]).

$$\mathrm{f_x}(X) = \frac{1}{b-1};\ (a < X < b) \tag{45}$$

Its properties are shown in the Equations 46.

$$\begin{aligned} \mathrm{E}(X) &= \frac{a+b}{2} \\ \mathrm{V}(X) &= \frac{(b-a)^2}{12} \\ \mathrm{u}(X) &= \frac{b-a}{\sqrt{12}} \end{aligned} \tag{46}$$

If the distribution is expressed using the half-width $\Delta a = \frac{b-a}{2}$ then the uncertainty is given by $\mathrm{u}(X) = \frac{\Delta a}{\sqrt{3}}$.

B.1.2. Triangular Distribution

The Triangular distribution, sometimes referred to as the Simpson distribution, is expressed by the PDF 47 of a random variable X that can take values in the interval $[a,b]$ only (*see* Wikipedia [45]).

$$\mathrm{f_x}(X) = \begin{cases} \frac{2(x-a)}{(b-a)(c-a)} & (a \leq x \leq c) \\ \frac{2(b-x)}{(b-a)(b-c)} & (c < x < b) \end{cases} \tag{47}$$

Its properties are shown in the Equations 48.

$$\begin{aligned} \mathrm{E}(X) &= \frac{a+b+c}{3} \\ \mathrm{V}(X) &= \frac{a^2+b^2+c^2-ab-ac-bc}{18} \\ \mathrm{u}(X) &= \sqrt{\mathrm{V}(X)} \end{aligned} \tag{48}$$

If $c = \frac{a+b}{2}$, then this distributions is also referred to as symmetric Triangular distribution. In this case the half-width is given by $\Delta a = \frac{b-a}{2}$ and $\mathrm{u}(X) = \frac{\Delta a}{\sqrt{6}}$.

B.1.3. Beta Distribution

The Beta distribution is expressed by the PDF 49 of a random variable X (*see* Wikipedia [46]).

$$\begin{aligned} \mathrm{f_x}(X) &= \frac{1}{\mathrm{B}(p,q)} x^{p-1}(1-x)^{q-1} \\ \mathrm{B}(p,q) &= \int_0^1 u^{p-1}(1-u)^{q-1}\,du \end{aligned} \tag{49}$$

Its properties are shown in the Equations 50.

$$\begin{aligned} \mathrm{E}(X) &= \frac{p}{p+q} \\ \mathrm{V}(X) &= \frac{pq}{(p+q+1)(p+q)^2} \\ \mathrm{u}(X) &= \sqrt{\mathrm{V}(X)} \end{aligned} \tag{50}$$

If $p = q = \frac{1}{2}$, then this distribution is also referred to as the Arcsine distribution.

B.1.4. Normal Distribution

The normal distribution is expressed by the PDF 51 of a random variable X (*see* Weise *et al* [4]).

$$\mathrm{f_x}(X) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{51}$$

Its properties are shown in the Equations 52.

$$\begin{aligned} \mathrm{E}(X) &= \mu \\ \mathrm{V}(X) &= \sigma^2 \\ \mathrm{u}(X) &= \sigma \end{aligned} \tag{52}$$

B.1.5. Multivariate Normal Distribution

Weise *et al* [4] provide an expression of a Normal PDF for a vector of random variables $\mathbf{X} = [x_1, x_2, \ldots, x_n]^T$.

$$\mathrm{f_x}(\mathbf{X}) = \frac{1}{2\pi\sqrt{\det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\mathbf{X}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{X}-\boldsymbol{\mu})\right) \tag{53}$$

Since this distribution describes the vector of random variables $\mathbf{X}$, it cannot be described in terms of a scalar expected value ($\mathrm{E}(X)$) and variance ($\mathrm{V}(X)$). Instead, the properties are an expected value vector ($\mathrm{E}(\mathbf{X})$) and a covariance matrix ($\mathrm{Cov}(\mathbf{X})$), shown in the Equations 54. The covariance matrix expresses the covariance of the components of the random vector. The standard uncertainty $\mathrm{u}(\mathbf{X})$ is the covariance matrix (see Section 4).

$$\begin{aligned} \mathrm{E}(\mathbf{X}) &= \boldsymbol{\mu} \\ \mathrm{Cov}(\mathbf{X}) &= \boldsymbol{\Sigma} \\ \mathrm{u}(\mathbf{X}) &= \boldsymbol{\Sigma} \end{aligned} \tag{54}$$

B.1.6. Bivariate Normal Distribution

Hall [18] simplifies the multivariate normal distribution to the bivariate case (*i.e.* $\mathbf{X} = [x_1, x_2]^T$). The bivariate normal distribution is expressed by the PDF 53 for a bivariate random vector $\mathbf{X} = [x_1, x_2]^T$. The evaluation of the expected value, the variance, and the uncertainty is done according to the multivariate normal distribution.

B.2. Monte-Carlo Integration

In this section we describe the approach of Monte-Carlo (MC) integration, proposed by Hüllermeier [28]. MC integration is an approximate approach to evaluate bounded multidimensional integrals. Consider the integral shown in Equation 55.

$$\boldsymbol{I} = \int_a^b h(x)dx \tag{55}$$

$$\boldsymbol{I} = \int_a^b w(x)f(x)dx \tag{56}$$

$$\begin{aligned} w(x) &= h(x)(b-a) \\ h(x) &= (b-a)^{-1} \end{aligned}$$

We can rewrite the integral shown in Equation 55 to the form shown in Equation 56. From the perspective probability theory this is the expected value of $w(x)$ of a uniformly distributed random variable x.

$$\hat{\boldsymbol{I}} = \frac{1}{N}\sum_{i=1}^{n} w(x_i) \tag{57}$$

$$\hat{se} = \frac{s}{\sqrt{n}} \tag{58}$$

$$s^2 = \frac{\sum_{i=1}^{n} \left(w(x_i) - \hat{\boldsymbol{I}}\right)^2}{n-1}$$

The expected value can be estimated by drawing n samples from a uniform distribution that has the parameters a and b. The estimation is performed using Equation 57. The standard error of the estimate is given by the Equation 58.

The major advantage of MC integration compared to other numerical integration procedures is that the standard error does not directly depend on the dimensionality of the problem. Other numerical integration procedures, such as the Runge-Kutta method (*see* Press *et al.* [47]), are usually applied iteratively on multidimensional integrals. Therefore the error directly depends on the dimensionality of the problem. One drawback of the MC method is that it converges slowly as implied by Equation 58.

B.3. The Central Limit Theorem

Let $X_1, X_2, \ldots$ be sequence of independent identically distributed random variables. Then the sum $Y_n = \sum_i X_i$ has the expected value $\mathrm{E}(Y_n) = n\mu$ and the variance $\mathrm{V}(Y_n) = n\sigma^2$. The distribution of the sum Y_n converges in distribution to the normal distribution $N(n\mu, n\sigma^2)$ as n approaches ∞, shown in Equation 59.

$$\begin{aligned} Z_n &= \frac{Y_n - n\mu}{\sqrt{n}\sigma} \\ \lim_{n\to\infty} \Pr(Z_n < z) &= \Phi(z) \end{aligned} \tag{59}$$

$\Phi(z)$ denotes the cummulative distribution function of the standard normal distribution [28].

B.4. Proof of the Equality of Both Approaches for Propagating Complex-Valued Uncertainty

In this Section we prove that both approaches of evaluating the uncertainty of complex-valued quantities (see Section 4) return equal results. From our perspective such a prove is necessary, since their equality is not obvious, and Hall's approach proposed in [7] is lacking a direct reference to [18].

It is to prove that the equations 60 and 61 are equal. For a detailed description of their components see Section 4.

$$\boldsymbol{V_y} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} \times \boldsymbol{V_x} \times \left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right)^T \tag{60}$$

$$\boldsymbol{V(y)} = \sum_{i=1}^{m} \sum_{i=j}^{m} \boldsymbol{U_i(y) R_{ij}(X) U_j(y)}^T \tag{61}$$

We begin by introducing the elements of the factors of equation 60. Since the matrix $\boldsymbol{V_x}$ is a covariance matrix, its components can be replaced using the relation of correlation coefficients and the covariance, shown in Equation 62. The value $\mathrm{u}(x_i)$ denotes the input uncertainty of the

respective input $\boldsymbol{x}(i)$ of the vector of the input quantities, denoted by $\boldsymbol{x}$.

$$\mathrm{cov}(x_i,x_j) = \mathrm{cov}(x_j,x_i) = \mathrm{u}(x_i)\,\mathrm{u}(x_j)\mathrm{r}(x_i,x_j)\; i = 1,2,\ldots,2m;\; j = 1,2,\ldots,2m \tag{62}$$

The value m is the number of complex-valued input quantities. The input quantities have already been decomposed into their respective real and imaginary parts, as discussed in Section 4. Thus the index $\boldsymbol{x}(i\cdot 2-1)$ represents the real part and the index $\boldsymbol{x}(i\cdot 2)$ expresses the imaginary part of the complex-valued input quantity $\boldsymbol{z_i}\; i = 1,2,..,m$ respectively. The matrix $\dfrac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$ expresses the Jacobian matrix of the model function with respect to all inputs. In the Equations 63, we show the components of the matrices we used in Equation 60.

$$\boldsymbol{V_x} = \begin{bmatrix} \mathrm{u}(x_1)\,\mathrm{u}(x_1)\mathrm{r}(x_1,x_1) & \mathrm{u}(x_1)\,\mathrm{u}(x_2)\mathrm{r}(x_1,x_2) & \ldots & \mathrm{u}(x_1)\,\mathrm{u}(x_{2m})\mathrm{r}(x_1,x_{2m}) \\ \mathrm{u}(x_2)\,\mathrm{u}(x_1)\mathrm{r}(x_2,x_1) & \mathrm{u}(x_2)\,\mathrm{u}(x_2)\mathrm{r}(x_2,x_2) & \ldots & \mathrm{u}(x_2)\,\mathrm{u}(x_{2m})\mathrm{r}(x_2,x_{2m}) \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{u}(x_{2m})\,\mathrm{u}(x_1)\mathrm{r}(x_{2m},x_1) & \mathrm{u}(x_{2m})\,\mathrm{u}(x_2)\mathrm{r}(x_{2m},x_2) & \ldots & \mathrm{u}(x_{2m})\,\mathrm{u}(x_{2m})\mathrm{r}(x_{2m},x_{2m}) \end{bmatrix}$$

$$\begin{aligned} \boldsymbol{f} &= f_1(\boldsymbol{x}) + j f_2(\boldsymbol{x}) \\ \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} &= \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_{2m}} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_{2m}} \end{bmatrix} \\ \left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right)^T &= \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_2}{\partial x_1} \\ \frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial x_2} \\ \vdots & \vdots \\ \frac{\partial f_1}{\partial x_{2m}} & \frac{\partial f_2}{\partial x_{2m}} \end{bmatrix} \end{aligned} \tag{63}$$

In order to keep the notation compact, we apply the following simplifications.

$$\begin{aligned} n &= 2m \\ \boldsymbol{V_x} &= \begin{bmatrix} c_{11} & c_{12} & \ldots & c_{1n} \\ c_{21} & c_{22} & \ldots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \ldots & c_{nn} \end{bmatrix} \\ c_{ij} &= \mathrm{u}(x_i)\,\mathrm{u}(x_j)\mathrm{r}(x_i,x_j);\; i = 1,2,\ldots,n;\; j = 1,2,\ldots,n \\ \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} &= \begin{bmatrix} y_{11} & y_{12} & \ldots & y_{1n} \\ y_{21} & y_{22} & \ldots & y_{2n} \end{bmatrix} \\ \left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right)^T &= \begin{bmatrix} y_{11} & y_{21} \\ y_{12} & y_{22} \\ \vdots & \vdots \\ y_{1n} & y_{2n} \end{bmatrix} \\ y_{1i} &= \frac{\partial f_1}{\partial x_i};\; i = 1,2,\ldots,n \end{aligned}$$

$$y_{2i} = \frac{\partial f_2}{\partial x_i};\ i = 1,2,\ldots,n$$

By the means of Linear Algebra (*i.e.* Lipschutz [48]) we obtain Equation 66 after the intermediate steps shown in the Equations 64 and 65.

$$\begin{aligned}
\boldsymbol{M_1} &= \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} \times \boldsymbol{V_x} \qquad (64)\\
&= \begin{bmatrix} y_{11} & y_{12} & \ldots & y_{1n} \\ y_{21} & y_{22} & \ldots & y_{2n} \end{bmatrix} \times \begin{bmatrix} c_{11} & c_{12} & \ldots & c_{1n} \\ c_{21} & c_{22} & \ldots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \ldots & c_{nn} \end{bmatrix} \\
&= \begin{bmatrix} \sum_{i=1}^{n} c_{i1} y_{1i} & \sum_{i=1}^{n} c_{i2} y_{1i} & \ldots & \sum_{i=1}^{n} c_{in} y_{1n} \\ \sum_{i=1}^{n} c_{i1} y_{2i} & \sum_{i=1}^{n} c_{i2} y_{2i} & \ldots & \sum_{i=1}^{n} c_{in} y_{2n} \end{bmatrix} \\
\boldsymbol{V_y} &= \boldsymbol{M_1} \times \left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} \right)^T \qquad (65)\\
&= \begin{bmatrix} \sum_{i=1}^{n} c_{i1} y_{1i} & \sum_{i=1}^{n} c_{i2} y_{1i} & \ldots & \sum_{i=1}^{n} c_{in} y_{1n} \\ \sum_{i=1}^{n} c_{i1} y_{2i} & \sum_{i=1}^{n} c_{i2} y_{2i} & \ldots & \sum_{i=1}^{n} c_{in} y_{2n} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{21} \\ y_{12} & y_{22} \\ \vdots & \vdots \\ y_{1n} & y_{2n} \end{bmatrix} \\
&= \begin{bmatrix} \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij} y_{1i} y_{1j} & \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij} y_{1i} y_{2j} \\ \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij} y_{2i} y_{1j} & \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij} y_{2i} y_{2j} \end{bmatrix} \\
&= \begin{bmatrix} \sum_{j=1}^{n} \sum_{i=1}^{n} \mathrm{r}(x_i,x_j)\,\mathrm{u}(x_i)\,\mathrm{u}(x_j) \frac{\partial f_1}{\partial x_i} \frac{\partial f_1}{\partial x_j} & \sum_{j=1}^{n} \sum_{i=1}^{n} \mathrm{r}(x_i,x_j)\,\mathrm{u}(x_i)\,\mathrm{u}(x_j) \frac{\partial f_1}{\partial x_i} \frac{\partial f_2}{\partial x_j} \\ \sum_{j=1}^{n} \sum_{i=1}^{n} \mathrm{r}(x_i,x_j)\,\mathrm{u}(x_i)\,\mathrm{u}(x_j) \frac{\partial f_2}{\partial x_i} \frac{\partial f_1}{\partial x_j} & \sum_{j=1}^{n} \sum_{i=1}^{n} \mathrm{r}(x_i,x_j)\,\mathrm{u}(x_i)\,\mathrm{u}(x_j) \frac{\partial f_2}{\partial x_i} \frac{\partial f_2}{\partial x_j} \end{bmatrix} \\
&= \sum_{i=1}^{2m} \sum_{j=1}^{2m} \begin{bmatrix} \frac{\partial f_1}{\partial x_i}\,\mathrm{u}(x_i)\mathrm{r}(x_i,x_j) \frac{\partial f_1}{\partial x_j}\,\mathrm{u}(x_j) & \frac{\partial f_1}{\partial x_i}\,\mathrm{u}(x_i)\mathrm{r}(x_i,x_j) \frac{\partial f_2}{\partial x_j}\,\mathrm{u}(x_j) \\ \frac{\partial f_2}{\partial x_i}\,\mathrm{u}(x_i)\mathrm{r}(x_i,x_j) \frac{\partial f_1}{\partial x_j}\,\mathrm{u}(x_j) & \frac{\partial f_2}{\partial x_i}\,\mathrm{u}(x_i)\mathrm{r}(x_i,x_j) \frac{\partial f_2}{\partial x_j}\,\mathrm{u}(x_j) \end{bmatrix} \qquad (66)
\end{aligned}$$

After presenting a formula how to calculate each element of the covariance matrix that expresses the combined standard uncertainty for the approach proposed by Hall in [18], we decompose equation 61 the same way. Hall [7] used another indexing scheme for the components of uncertainty in this approach. The value x_{1i} expresses the real part and the value x_{2i} expresses the imaginary part of the complex-valued input quantity $\boldsymbol{z_i}$. We will later apply a transformation to this indexing scheme in order to make it comparable to the scheme used for the approach proposed by Hall in [18]. The matrix $\boldsymbol{U_j(y)}$ expresses the Jacobian matrix of the model Function $\boldsymbol{f}$ with respect to the complex-valued input quantity $\boldsymbol{z_j}$ exclusively. The matrix $\mathrm{u}(\boldsymbol{z_j})$ contains the input uncertainties of the complex-valued input quantity $\boldsymbol{z_j}$, as shown in Equation 67. The matrix $\boldsymbol{R_{ij}(X)}$ contains the correlation coefficients of the real and imaginary parts of the complex-valued input quantities $\boldsymbol{z_i}$ and $\boldsymbol{z_j}$, as shown in Equation 68.

$$\boldsymbol{U_j(y)} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z_j}} \times \mathrm{u}(\boldsymbol{z_j})$$

$$= \begin{bmatrix} \frac{\partial f_1}{\partial x_{1j}} & \frac{\partial f_1}{\partial x_{2j}} \\ \frac{\partial f_2}{\partial x_{1j}} & \frac{\partial f_2}{\partial x_{2j}} \end{bmatrix} \times \begin{bmatrix} \mathrm{u}(x_1 j) & 0 \\ 0 & \mathrm{u}(x_2 j) \end{bmatrix} \tag{67}$$

$$= \begin{bmatrix} \frac{\partial f_1}{\partial x_{1j}} \mathrm{u}(x_{1j}) & \frac{\partial f_1}{\partial x_{2j}} \mathrm{u}(x_{2j}) \\ \frac{\partial f_2}{\partial x_{1j}} \mathrm{u}(x_{1j}) & \frac{\partial f_2}{\partial x_{2j}} \mathrm{u}(x_{2j}) \end{bmatrix}$$

$$\boldsymbol{R}_{ij}(\boldsymbol{X}) = \begin{bmatrix} \mathrm{r}(x_{1i}, x_{1j}) & \mathrm{r}(x_{1i}, x_{2j}) \\ \mathrm{r}(x_{2i}, x_{2j}) & \mathrm{r}(x_{2i}, x_{2j}) \end{bmatrix} \tag{68}$$

By the means of Linear Algebra (*i.e.* Lipschutz [48]) we obtain Equation 72 after the intermediate steps shown in the Equations 69 and 70.

$$\boldsymbol{N_1} = \boldsymbol{R}_{ij}(\boldsymbol{X}) \times \boldsymbol{U_j}(\boldsymbol{Y})^T \tag{69}$$

$$= \begin{bmatrix} \mathrm{r}(x_{1i}, x_{1j}) & \mathrm{r}(x_{1i}, x_{2j}) \\ \mathrm{r}(x_{2i}, x_{2j}) & \mathrm{r}(x_{2i}, x_{2j}) \end{bmatrix} \times \begin{bmatrix} \frac{\partial f_1}{\partial x_{1j}} & \frac{\partial f_2}{\partial x_{1j}} \\ \frac{\partial f_1}{\partial x_{2j}} & \frac{\partial f_2}{\partial x_{2j}} \end{bmatrix}$$

$$= \begin{bmatrix} \sum_{k=1}^{2} \mathrm{r}(x_{1i}, x_{kj}) \frac{\partial f_1}{\partial x_{kj}} & \sum_{k=1}^{2} \mathrm{r}(x_{1i}, x_{kj}) \frac{\partial f_2}{\partial x_{kj}} \\ \sum_{k=1}^{2} \mathrm{r}(x_{2i}, x_{kj}) \frac{\partial f_1}{\partial x_{kj}} & \sum_{k=1}^{2} \mathrm{r}(x_{2i}, x_{kj}) \frac{\partial f_2}{\partial x_{kj}} \end{bmatrix}$$

$$\boldsymbol{N_2} = \boldsymbol{U_i}(\boldsymbol{Y}) \times \boldsymbol{N_1} \tag{70}$$

$$\tag{71}$$

$$= \begin{bmatrix} \sum_{h=1}^{2} \sum_{k=1}^{2} \frac{\partial f_1}{\partial x_{hi}} \mathrm{u}(x_{hi}) \mathrm{r}(x_{hi}, x_{kj}) \frac{\partial f_1}{\partial x_{kj}} \mathrm{u}(x_{kj}) & \sum_{h=1}^{2} \sum_{k=1}^{2} \frac{\partial f_1}{\partial x_{hi}} \mathrm{u}(x_{hi}) \mathrm{r}(x_{hi}, x_{kj}) \frac{\partial f_2}{\partial x_{kj}} \mathrm{u}(x_{kj}) \\ \sum_{h=1}^{2} \sum_{k=1}^{2} \frac{\partial f_2}{\partial x_{hi}} \mathrm{u}(x_{hi}) \mathrm{r}(x_{hi}, x_{kj}) \frac{\partial f_1}{\partial x_{kj}} \mathrm{u}(x_{kj}) & \sum_{h=1}^{2} \sum_{k=1}^{2} \frac{\partial f_2}{\partial x_{hi}} \mathrm{u}(x_{hi}) \mathrm{r}(x_{hi}, x_{kj}) \frac{\partial f_2}{\partial x_{kj}} \mathrm{u}(x_{kj}) \end{bmatrix}$$

$$\boldsymbol{V}(\boldsymbol{y}) = \sum_{i=1}^{m} \sum_{j=1}^{m} \sum_{h=1}^{2} \sum_{k=1}^{2} \begin{bmatrix} \frac{\partial f_1}{\partial x_{hi}} \mathrm{u}(x_{hi}) \mathrm{r}(x_{hi}, x_{kj}) \frac{\partial f_1}{\partial x_{kj}} \mathrm{u}(x_{kj}) & \frac{\partial f_1}{\partial x_{hi}} \mathrm{u}(x_{hi}) \mathrm{r}(x_{hi}, x_{kj}) \frac{\partial f_2}{\partial x_{kj}} \mathrm{u}(x_{kj}) \\ \frac{\partial f_2}{\partial x_{hi}} \mathrm{u}(x_{hi}) \mathrm{r}(x_{hi}, x_{kj}) \frac{\partial f_1}{\partial x_{kj}} \mathrm{u}(x_{kj}) & \frac{\partial f_2}{\partial x_{hi}} \mathrm{u}(x_{hi}) \mathrm{r}(x_{hi}, x_{kj}) \frac{\partial f_2}{\partial x_{kj}} \mathrm{u}(x_{kj}) \end{bmatrix} \tag{72}$$

Since $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, m$, we can apply the index transformation 73 and obtain the same result as shown in Equation 66. Therefore both approaches to propagate the uncertainty of complex-valued uncertainties proposed by Hall [7, 18] return equal results.

$$\begin{aligned} \sum_{i=1}^{2m} &:= \sum_{i=1}^{m} \sum_{k=1}^{2} \\ \sum_{j=1}^{2m} &:= \sum_{j=1}^{m} \sum_{h=1}^{2} \\ x_i &:= x_{hi} \\ x_j &:= x_{kj} \end{aligned} \tag{73}$$

$$\boldsymbol{V}(\boldsymbol{y}) = \sum_{i=1}^{2m} \sum_{j=1}^{2m} \begin{bmatrix} \frac{\partial f_1}{\partial x_i} \mathrm{u}(x_i) \mathrm{r}(x_i, x_j) \frac{\partial f_1}{\partial x_j} \mathrm{u}(x_j) & \frac{\partial f_1}{\partial x_i} \mathrm{u}(x_i) \mathrm{r}(x_i, x_j) \frac{\partial f_2}{\partial x_j} \mathrm{u}(x_j) \\ \frac{\partial f_2}{\partial x_i} \mathrm{u}(x_i) \mathrm{r}(x_i, x_j) \frac{\partial f_1}{\partial x_j} \mathrm{u}(x_j) & \frac{\partial f_2}{\partial x_i} \mathrm{u}(x_i) \mathrm{r}(x_i, x_j) \frac{\partial f_2}{\partial x_j} \mathrm{u}(x_j) \end{bmatrix} \tag{74}$$

Quod erat demonstrandum.

B.5. Complex Differentiable Functions and the Cauchy-Riemann Equations

In this section we describe the Cauchy-Riemann equations and how they can be used to obtain the Jacobian matrix for complex-valued functions. According to Wolfram Research [49], a complex function, shown in Equation 75), is called complex differentiable on some region $\mathbb{G}$ containing the point $\boldsymbol{z_0}$ if and only if $\boldsymbol{f}(\boldsymbol{z})$ has continuous first partial derivates and satisfies the Cauchy-Riemann equations (see Equations 76).

$$\begin{aligned} \boldsymbol{f}(\boldsymbol{z}) &= u(x,y) + jv(x,y);\ \boldsymbol{z} = a + jb && (75) \\ \frac{\partial u}{\partial x} &= \frac{\partial v}{\partial y} \\ \frac{\partial v}{\partial x} &= -\frac{\partial u}{\partial y} && (76) \end{aligned}$$

The derivate is then given by Equation 77.

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}(\boldsymbol{z_0}) = \lim_{\boldsymbol{z} \to \boldsymbol{z_0}} \frac{\boldsymbol{f}(\boldsymbol{z}) - \boldsymbol{f}(\boldsymbol{z_0})}{\boldsymbol{z} - \boldsymbol{z_0}} \tag{77}$$

A function is complex differentiable if and only if its Jacobian matrix is of the Form 78.

$$\boldsymbol{J} = \begin{bmatrix} A & -B \\ B & A \end{bmatrix} \tag{78}$$

According to Hall [18], this principle can be extended to functions having multiple complex-valued input parameters, such as shown in Equation 79.

$$\boldsymbol{f}(\boldsymbol{z_1}, \boldsymbol{z_2}, \ldots, \boldsymbol{z_n}) = u(\boldsymbol{z_1}, \boldsymbol{z_2}, \ldots, \boldsymbol{z_n}) + jv(\boldsymbol{z_1}, \boldsymbol{z_2}, \ldots, \boldsymbol{z_n}) \tag{79}$$

In this case, the complex Jacobian matrix is obtained first, shown in Equation 80. After that, the elements of the complex Jacobian matrix are transformed using the Transformation 81, obtaining the Jacobian matrix, shown in Equation 82. For the equations below let $\boldsymbol{z_i} = x_i + jy_i$.

$$\begin{aligned} \boldsymbol{J_z} &= \begin{bmatrix} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z_1}} & \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z_2}} & \cdots & \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z_n}} \end{bmatrix} && (80) \\ \mathrm{M}(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z_i}}) &= \begin{bmatrix} \frac{\partial u}{\partial x_i} & -\frac{\partial v}{\partial y_i} \\ \frac{\partial v}{\partial y_i} & \frac{\partial u}{\partial x_i} \end{bmatrix} && (81) \\ \boldsymbol{J} &= \begin{bmatrix} \frac{\partial u}{\partial x_1} & -\frac{\partial v}{\partial y_1} & \frac{\partial u}{\partial x_2} & -\frac{\partial v}{\partial y_2} & \cdots & \frac{\partial u}{\partial x_n} & -\frac{\partial v}{\partial y_n} \\ \frac{\partial v}{\partial y_1} & \frac{\partial u}{\partial x_1} & \frac{\partial v}{\partial y_2} & \frac{\partial u}{\partial x_2} & \cdots & \frac{\partial v}{\partial y_n} & \frac{\partial u}{\partial x_n} \end{bmatrix} && (82) \end{aligned}$$

B.6. Derivation of Selected Complex-Valued Functions

In this section, we derive the Jacobian Matrices for the complex-valued functions that are implemented in our class library. These matrices are crucial for calculating the uncertainty of complex valued input parameters, as discussed in Section 4. The general approach to derive Jacobian matrices of complex valued functions is described below.

1. The complex characteristics are calculated. That is deriving a formula for the real Part $\Re(f)$ and one formula for the imaginary Part $\Im(f)$ of the respective Function f.
2. Partial derivation is applied to the complex Characteristics.
3. The partial derivates are placed into the Jacobian matrix $\mathrm{J}(f)$, shown in Equation 83.

$$\mathrm{J}(f) = \begin{bmatrix} \frac{\partial}{\partial x}\Re(f) & \frac{\partial}{\partial y}\Re(f) \\ \frac{\partial}{\partial x}\Im(f) & \frac{\partial}{\partial y}\Im(f) \end{bmatrix} \tag{83}$$

Hall [18] describes another approach for complex valued functions that fulfil the Cauchy-Riemann equations, described in Section B.5. A function that fulfils this criterion is also referred to as analytic or holomorphic. His approach can be divided into the following intermediate steps.

1. Check the function for analyticity.
2. Evaluate the complex Jacobian matrix J_z, shown in Equation 84.
3. Transform the complex Jacobian matrix J_z into a scalar Jacobian matrix J, shown in Equation 85.

Because of its complexity, we do not evaluate Point 1 explicitly in this Section for the functions we used. Instead, we cite an appropriate source where the analyticity has already been evaluated.

$$\begin{aligned} f(\boldsymbol{z_1}, \boldsymbol{z_2}, \ldots, \boldsymbol{z_m}) &= u(x_1, y_1, x_2, y_2, \ldots, x_m, y_m) + jv(x_1, y_1, x_2, y_2, \ldots, x_m, y_m) \\ \mathrm{J}_z(f) &= \begin{bmatrix} \frac{\partial f}{\partial \boldsymbol{z_1}} & \frac{\partial f}{\partial \boldsymbol{z_2}} & \cdots & \frac{\partial f}{\partial \boldsymbol{z_m}} \end{bmatrix} \end{aligned} \tag{84}$$

$$\mathrm{M}(\mathrm{J}_z(f)) = \mathrm{J}(f) = \begin{bmatrix} \frac{\partial u}{\partial x_1} & -\frac{\partial v}{\partial y_1} & \frac{\partial u}{\partial x_2} & -\frac{\partial v}{\partial y_2} & \cdots & \frac{\partial u}{\partial x_m} & -\frac{\partial v}{\partial y_m} \\ \frac{\partial v}{\partial y_1} & \frac{\partial u}{\partial x_1} & \frac{\partial v}{\partial y_2} & \frac{\partial u}{\partial x_2} & \cdots & \frac{\partial v}{\partial y_m} & \frac{\partial u}{\partial x_m} \end{bmatrix} \tag{85}$$

B.6.1. Absolute Value

Properties	
Operator:	$\lvert\boldsymbol{z}\rvert$
Domain :	Defined over the entire complex plane
Analyticity:	Not analytical, as seen in Equations 86.

$$\begin{aligned} |\boldsymbol{z}| = \sqrt{x^2+y^2} \quad &: \quad \mathbb{C} \to \mathbb{R} \\ \frac{\partial}{\partial x}|\boldsymbol{z}| &= \frac{x}{\sqrt{x^2+y^2}} \\ \frac{\partial}{\partial y}|\boldsymbol{z}| &= \frac{y}{\sqrt{x^2+y^2}} \end{aligned}$$

$$\boldsymbol{J}(|\boldsymbol{z}|) = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} \\ 0 & 0 \end{bmatrix} \tag{86}$$

B.6.2. Complex Conjugate

Properties	
Operator:	$\bar{z}$
Domain :	Defined over the entire complex plane
Analyticity:	Not analytical, as seen in Equations 87.

$$\begin{aligned} \bar{z} = \overline{x+iy} = x - iy \quad &: \quad \mathbb{C} \to \mathbb{C} \\ \Re(\bar{z}) &= x \\ \Im(\bar{z}) &= -y \\ \frac{\partial}{\partial x}\Re(\bar{z}) &= 1 \\ \frac{\partial}{\partial y}\Re(\bar{z}) &= 0 \\ \frac{\partial}{\partial x}\Im(\bar{z}) &= 0 \\ \frac{\partial}{\partial y}\Im(\bar{z}) &= -1 \\ \boldsymbol{J}(\bar{z}) &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \end{aligned} \tag{87}$$

B.6.3. Negation

Properties	
Operator:	$\bar{z}$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, as seen in Equations 88.

$$\begin{aligned} -\boldsymbol{z} = -(x+iy) = -x - iy \quad &: \quad \mathbb{C} \to \mathbb{C} \\ \Re(-\boldsymbol{z}) &= -x \\ \Im(-\boldsymbol{z}) &= -y \\ \frac{\partial}{\partial x}\Re(-\boldsymbol{z}) &= -1 \\ \frac{\partial}{\partial y}\Re(-\boldsymbol{z}) &= 0 \\ \frac{\partial}{\partial x}\Im(-\boldsymbol{z}) &= 0 \\ \frac{\partial}{\partial y}\Im(-\boldsymbol{z}) &= -1 \end{aligned}$$

$$\boldsymbol{J}(-\boldsymbol{z}) = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \tag{88}$$

B.6.4. Inversion

Properties	
Operator:	z^{-1}
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, as seen in Equations 89.

$$\begin{aligned} \boldsymbol{z}^{-1} = \frac{1}{z} &: \mathbb{C} \to \mathbb{C} \\ \frac{\partial}{\partial \boldsymbol{z}} \boldsymbol{z}^{-1} &= -\frac{1}{\boldsymbol{z}^2} \\ \boldsymbol{J}(\boldsymbol{z}^{-1}) &= \begin{bmatrix} \Re(-\frac{1}{\boldsymbol{z}^2}) & -\Im(-\frac{1}{\boldsymbol{z}^2}) \\ \Im(-\frac{1}{\boldsymbol{z}^2}) & \Re(-\frac{1}{\boldsymbol{z}^2}) \end{bmatrix} \end{aligned} \tag{89}$$

B.6.5. Square-Root

Properties	
Operator:	$\sqrt{z}$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \sqrt{\boldsymbol{z}} &: \mathbb{C} \to \mathbb{C} \\ \frac{\partial}{\partial \boldsymbol{z}} \sqrt{\boldsymbol{z}} &= \frac{1}{2\sqrt{\boldsymbol{z}}} \\ \boldsymbol{J}(\sqrt{\boldsymbol{z}}) &= \begin{bmatrix} \Re(\frac{1}{2\sqrt{\boldsymbol{z}}}) & -\Im(\frac{1}{2\sqrt{\boldsymbol{z}}}) \\ \Im(\frac{1}{2\sqrt{\boldsymbol{z}}}) & \Re(\frac{1}{2\sqrt{\boldsymbol{z}}}) \end{bmatrix} \end{aligned} \tag{90}$$

B.6.6. Exponential Function

Properties	
Operator:	$\exp(\boldsymbol{z})$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \exp(\boldsymbol{z}) &: \mathbb{C} \to \mathbb{C} \\ \frac{\partial}{\partial \boldsymbol{z}} \exp(z) &= \exp(\boldsymbol{z}) \\ \boldsymbol{J}(\exp(\boldsymbol{z})) &= \begin{bmatrix} \Re(\exp(\boldsymbol{z})) & -\Im(\exp(\boldsymbol{z})) \\ \Im(\exp(\boldsymbol{z})) & \Re(\exp(\boldsymbol{z})) \end{bmatrix} \end{aligned} \tag{91}$$

B.6.7. Natural Logarithm

Properties	
Operator:	ln
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \ln(\boldsymbol{z}) &: \mathbb{C} \to \mathbb{C} \\ \frac{\partial}{\partial \boldsymbol{z}} \ln(\boldsymbol{z}) &= \frac{1}{\boldsymbol{z}} \\ \boldsymbol{J}(\ln(\boldsymbol{z})) &= \begin{bmatrix} \Re(\frac{1}{\boldsymbol{z}}) & -\Im(\frac{1}{\boldsymbol{z}}) \\ \Im(\frac{1}{\boldsymbol{z}}) & \Re(\frac{1}{\boldsymbol{z}}) \end{bmatrix} \end{aligned} \tag{92}$$

B.6.8. Sine Function

Properties	
Operator:	$\sin(\boldsymbol{z})$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \sin(\boldsymbol{z}) &: \mathbb{C} \to \mathbb{C} \\ \frac{\partial}{\partial \boldsymbol{z}} \sin(\boldsymbol{z}) &= \cos(\boldsymbol{z}) \\ \boldsymbol{J}(\sin(\boldsymbol{z})) &= \begin{bmatrix} \Re(\cos(\boldsymbol{z})) & -\Im(\cos(\boldsymbol{z})) \\ \Im(\cos(\boldsymbol{z})) & \Re(\cos(\boldsymbol{z})) \end{bmatrix} \end{aligned} \tag{93}$$

B.6.9. Cosine Function

Properties	
Operator:	$\cos(\boldsymbol{z})$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \cos(\boldsymbol{z}) &: \mathbb{C} \to \mathbb{C} \\ \frac{\partial}{\partial \boldsymbol{z}} \cos(\boldsymbol{z}) &= -\sin(\boldsymbol{z}) \\ \boldsymbol{J}(\cos(\boldsymbol{z})) &= \begin{bmatrix} \Re(-\sin(\boldsymbol{z})) & -\Im(-\sin(\boldsymbol{z})) \\ \Im(-\sin(\boldsymbol{z})) & \Re(-\sin(\boldsymbol{z})) \end{bmatrix} \end{aligned} \tag{94}$$

B.6.10. Tangent Function

Properties	
Operator:	$\tan(z)$
Domain :	Defined over the entire complex plane, except for the values shown in Equation 95
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \tan(z) &: \mathbb{C} \to \mathbb{C};\ z \neq \frac{\pi}{2} + k\pi;\ k \in \mathbb{Z} \\ \frac{\partial}{\partial z}\tan(z) &= \frac{1}{\cos^2(z)} \\ \boldsymbol{J}(\tan(z)) &= \begin{bmatrix} \Re(\frac{1}{\cos^2(z)}) & -\Im(\frac{1}{\cos^2(z)}) \\ \Im(\frac{1}{\cos^2(z)}) & \Re(\frac{1}{\cos^2(z)}) \end{bmatrix} \end{aligned} \tag{95}$$

B.6.11. Inverse Sine Function (Arc-Sine Function)

Properties	
Operator:	$\sin^{-1}(z)$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \sin^{-1}(z) &: \mathbb{C} \to \mathbb{C} \\ \frac{\partial}{\partial z}\sin^{-1}(z) &= \frac{1}{\sqrt{1-z^2}} \\ \boldsymbol{J}(\sin^{-1}(z)) &= \begin{bmatrix} \Re(\frac{1}{\sqrt{1-z^2}}) & -\Im(\frac{1}{\sqrt{1-z^2}}) \\ \Im(\frac{1}{\sqrt{1-z^2}}) & \Re(\frac{1}{\sqrt{1-z^2}}) \end{bmatrix} \end{aligned} \tag{96}$$

B.6.12. Inverse Cosine Function (Arc-Cosine Function)

Properties	
Operator:	$\cos^{-1}(z)$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \cos^{-1}(z) &: \mathbb{C} \to \mathbb{C} \\ \frac{\partial}{\partial z}\cos^{-1}(z) &= -\frac{1}{\sqrt{1-z^2}} \\ \boldsymbol{J}(\cos^{-1}(z)) &= \begin{bmatrix} \Re(-\frac{1}{\sqrt{1-z^2}}) & -\Im(-\frac{1}{\sqrt{1-z^2}}) \\ \Im(-\frac{1}{\sqrt{1-z^2}}) & \Re(-\frac{1}{\sqrt{1-z^2}}) \end{bmatrix} \end{aligned} \tag{97}$$

B.6.13. Inverse Tangent Function (Arc-Tangent Function)

Properties	
Operator:	$\tan^{-1}(\boldsymbol{z})$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \tan^{-1}(\boldsymbol{z}) &: \mathbb{C} \rightarrow \mathbb{C} \\ \frac{\partial}{\partial \boldsymbol{z}} \tan^{-1}(\boldsymbol{z}) &= \frac{1}{\boldsymbol{z}^2+1} \\ \boldsymbol{J}(\tan^{-1}(\boldsymbol{z})) &= \begin{bmatrix} \Re(\frac{1}{\boldsymbol{z}^2+1}) & -\Im(\frac{1}{\boldsymbol{z}^2+1}) \\ \Im(\frac{1}{\boldsymbol{z}^2+1}) & \Re(\frac{1}{\boldsymbol{z}^2+1}) \end{bmatrix} \end{aligned} \tag{98}$$

B.6.14. Hyperbolic Sine Function

Properties	
Operator:	$\sinh(\boldsymbol{z})$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \sinh(\boldsymbol{z}) &: \mathbb{C} \rightarrow \mathbb{C} \\ \frac{\partial}{\partial \boldsymbol{z}} \sinh(\boldsymbol{z}) &= \cosh(\boldsymbol{z}) \\ \boldsymbol{J}(\sinh(\boldsymbol{z})) &= \begin{bmatrix} \Re(\cosh(\boldsymbol{z})) & -\Im(\cosh(\boldsymbol{z})) \\ \Im(\cosh(\boldsymbol{z})) & \Re(\cosh(\boldsymbol{z})) \end{bmatrix} \end{aligned} \tag{99}$$

B.6.15. Hyperbolic Cosine Function

Properties	
Operator:	$\cosh(\boldsymbol{z})$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \cosh(\boldsymbol{z}) &: \mathbb{C} \rightarrow \mathbb{C} \\ \frac{\partial}{\partial \boldsymbol{z}} \cosh(\boldsymbol{z}) &= \sinh(\boldsymbol{z}) \\ \boldsymbol{J}(\cosh(\boldsymbol{z})) &= \begin{bmatrix} \Re(\sinh(\boldsymbol{z})) & -\Im(\sinh(\boldsymbol{z})) \\ \Im(\sinh(\boldsymbol{z})) & \Re(\sinh(\boldsymbol{z})) \end{bmatrix} \end{aligned} \tag{100}$$

B.6.16. Hyperbolic Tangent Function

Properties	
Operator:	$\tanh(z)$
Domain :	Defined over the entire complex plane, except for the values shown in Equation 101
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \tanh(z) &: \mathbb{C} \to \mathbb{C};\ z \neq \frac{\pi i}{2} + k\pi i;\ k \in \mathbb{Z} \\ \frac{\partial}{\partial z}\tanh(z) &= \frac{1}{\cosh^2(z)} \\ \boldsymbol{J}(\tanh(z)) &= \begin{bmatrix} \Re(\frac{1}{\cosh^2(z)}) & -\Im(\frac{1}{\cosh^2(z)}) \\ \Im(\frac{1}{\cosh^2(z)}) & \Re(\frac{1}{\cosh^2(z)}) \end{bmatrix} \end{aligned} \tag{101}$$

B.6.17. Inverse Hyperbolic Sine Function

Properties	
Operator:	$\sinh^{-1}(z)$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \sinh^{-1}(z) &: \mathbb{C} \to \mathbb{C} \\ \frac{\partial}{\partial z}\sinh^{-1}(z) &= \frac{1}{\sqrt{1+z^2}} \\ \boldsymbol{J}(\sinh^{-1}(z)) &= \begin{bmatrix} \Re(\frac{1}{\sqrt{1+z^2}}) & -\Im(\frac{1}{\sqrt{1+z^2}}) \\ \Im(\frac{1}{\sqrt{1+z^2}}) & \Re(\frac{1}{\sqrt{1+z^2}}) \end{bmatrix} \end{aligned} \tag{102}$$

B.6.18. Inverse Hyperbolic Cosine Function

Properties	
Operator:	$\cosh^{-1}(z)$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \cosh^{-1}(z) &: \mathbb{C} \to \mathbb{C} \\ \frac{\partial}{\partial z}\cosh^{-1}(z) &= \frac{1}{\sqrt{z-1}\sqrt{z+1}} \\ \boldsymbol{J}(\cosh^{-1}(z)) &= \begin{bmatrix} \Re(\frac{1}{\sqrt{z-1}\sqrt{z+1}}) & -\Im(\frac{1}{\sqrt{z-1}\sqrt{z+1}}) \\ \Im(\frac{1}{\sqrt{z-1}\sqrt{z+1}}) & \Re(\frac{1}{\sqrt{z-1}\sqrt{z+1}}) \end{bmatrix} \end{aligned} \tag{103}$$

B.6.19. Inverse Hyperbolic Tangent Function

Properties	
Operator:	$\tanh^{-1}(z)$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$\begin{aligned} \tanh^{-1}(z) &: \mathbb{C} \to \mathbb{C} \\ \frac{\partial}{\partial z}\tanh^{-1}(z) &= \frac{1}{1-z^2} \\ \boldsymbol{J}(\tanh^{-1}(z)) &= \begin{bmatrix} \Re(\frac{1}{1-z^2}) & -\Im(\frac{1}{1-z^2}) \\ \Im(\frac{1}{1-z^2}) & \Re(\frac{1}{1-z^2}) \end{bmatrix} \end{aligned} \tag{104}$$

B.6.20. Complex Addition

Properties	
Operator:	$z_1 + z_2$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical

$$\begin{aligned} z^a &: \mathbb{C} \times \mathbb{C} \to \mathbb{C} \\ \frac{\partial}{\partial z_1} z_1 + z_2 &= 1 \\ \frac{\partial}{\partial z_2} z_1 + z_2 &= 1 \\ \boldsymbol{J}(z_1 + z_2) &= \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \end{aligned} \tag{105}$$

B.6.21. Complex Multiplication

Properties	
Operator:	$z_1 z_2$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to Hall [18].

$$\begin{aligned} z_1 z_2 &: \mathbb{C} \times \mathbb{C} \to \mathbb{C} \\ \frac{\partial}{\partial z_1} z_1 z_2 &= z_2 \\ \frac{\partial}{\partial z_2} z_1 z_2 &= z_1 \\ \boldsymbol{J}(z_1 z_2) &= \begin{bmatrix} \Re(z_2) & -\Im(z_2) & \Re(z_1) & -\Im(z_1) \\ \Im(z_2) & \Re(z_2) & \Im(z_1) & \Re(z_1) \end{bmatrix} \end{aligned} \tag{106}$$

B.6.22. Complex Division

Properties	
Operator:	$z_1 z_2^{-1}$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical

$$
\begin{aligned}
z_1 z_2^{-1} &: \mathbb{C} \times \mathbb{C} \to \mathbb{C} \\
\frac{\partial}{z_1} z_1 z_2^{-1} &= z_2^{-1} \\
\frac{\partial}{z_2} z_1 z_2^{-1} &= -z_1 z_2^{-2} \\
\boldsymbol{J}(z_1 + z_2) &= \begin{bmatrix} \Re(z_2^{-1}) & -\Im(z_2^{-1}) & \Re(-z_1 z_2^{-2}) & -\Im(-z_1 z_2^{-2}) \\ \Im(z_2^{-1}) & \Re(z_2^{-1}) & \Im(-z_1 z_2^{-2}) & \Re(-z_1 z_2^{-2}) \end{bmatrix}
\end{aligned} \tag{107}
$$

B.6.23. Complex Powers

Properties	
Operator:	z^a
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$
\begin{aligned}
z^a &: \mathbb{C} \times \mathbb{C} \to \mathbb{C} \\
\frac{\partial}{\partial z} z^a &= a z^{a-1} \\
\frac{\partial}{\partial a} z^a &= z^a \ln(z) \\
\boldsymbol{J}(z^a) &= \begin{bmatrix} \Re(a z^{a-1}) & -\Im(a z^{a-1}) & \Re(z^a \ln(z)) & -\Im(z^a \ln(z)) \\ \Im(a z^{a-1}) & \Re(a z^{a-1}) & \Im(z^a \ln(z)) & \Re(z^a \ln(z)) \end{bmatrix}
\end{aligned} \tag{108}
$$

B.6.24. Inverse Two-Argument Tangent Function

Properties	
Operator:	$\tan_2^{-1}(x, y)$
Domain :	Defined over the entire complex plane
Analyticity:	Analytical, according to the Wolfram Function Site [50].

$$
\begin{aligned}
\tan_2^{-1} &: \mathbb{C} \times \mathbb{C} \to \mathbb{C} \\
\frac{\partial}{\partial x} \tan_2^{-1}(x, y) &= \frac{y}{x^2 + y^2} \\
\frac{\partial}{\partial y} \tan_2^{-1}(x, y) &= \frac{x}{x^2 + y^2}
\end{aligned}
$$

$$\boldsymbol{J}(\boldsymbol{z^a}) \quad = \quad \begin{bmatrix} \Re(\frac{\boldsymbol{y}}{\boldsymbol{x^2+y^2}}) & -\Im(\frac{\boldsymbol{y}}{\boldsymbol{x^2+y^2}}) & \Re(\frac{\boldsymbol{x}}{\boldsymbol{x^2+y^2}}) & -\Im(\frac{\boldsymbol{x}}{\boldsymbol{x^2+y^2}}) \\ \Im(\frac{\boldsymbol{y}}{\boldsymbol{x^2+y^2}}) & \Re(\frac{\boldsymbol{y}}{\boldsymbol{x^2+y^2}}) & \Im(\frac{\boldsymbol{x}}{\boldsymbol{x^2+y^2}}) & \Re(\frac{\boldsymbol{x}}{\boldsymbol{x^2+y^2}}) \end{bmatrix} \tag{109}$$

C. SCUQ Programming Manual

In this section we included the SCUQ programming manual. It includes a complete interface description of all classes and all functions of SCUQ. Furthermore, it documents the following:

- Installation and verification instructions.
- A documentation of all modules and files of the SCUQ software distribution.
- More examples demonstrating the use of SCUQ.
- Coercion rules of the data types implemented.

In order to save space, we packed two pages of the programming manual onto one page of this section. This programming manual can also be created from the source code of SCUQ in HTML- or PDF format, which we describe later this section.

Contents

1 SCUQ - A Class Library for the Evaluation of Scalar- and Complex-valued Uncertain Quantities.

This class library supports the evaluation of scalar (real) and complex-valued uncertain quantities. We divided the the library into the following modules.

- The module **scuq.units** (p. 35) supports modeling and converting physical units.
- The module **scuq.si** (p. 30) uses the units module to support SI units.
- The module **scuq.arithmetic** (p. 25). This module contains functions to assist the other modules in this libary. It also contains a RationalNumber type according to PEP-239 (see link shown below).
- The module **scuq.quantities** (p. 29) allows combining units, numeric types, and uncertain components modeling physical quantities.
- The module **scuq.ucomponents** (p. 33) module models uncertain values. It can be used in combination with the other modules to model uncertainty in measurements by assigning an uncertainty to a numeric value and propagating it through a mathematical model. The implementation uses the GUM-Tree pattern (see references shown below).
- The module **scuq.cucomponents** (p. 26) can be used to evaluate the uncertainty of complex-valued models in a similar way as the module **scuq.ucomponents** (p. 33) does.

Attention:

In contrast to the practice of explicit type checking and raising a `TypeError` if an argument is invalid, we use assertions. This gives you the opportunity to

check your assignments in debug mode and running (relatively) fast code in release mode. The debug mode is enabled by default when invoking Python with `python <Your Code>` and `python -O <Your Code>` for release mode.
You should use `UTF-8` as default encoding because Greek letters represent some physical quantities, units, and dimensions. However, you will still be able to use this library if you have another default encoding. The symbols will then not print correctly.
In this documentation the term `integer` refers to they Python type `int` as well as `long`. This library casts all `int` arguments to `long` where applicable. This makes overflows unlikely, since the precision of `long` is limited by the platforms available memory in Python; that said, you will most likely encounter a `MemoryError` if the accuracy of a long variable is exausted.

Note:

The patterns used to create the units, dimensions, and unit-operators have been inspired by Java Specification Request 275 that is implemented in `JScience` (see link shown below), an open-source library for scientific computing in Java.
The design patterns used for the evaluation of uncertainty are subject to United States patent number 7,130,761. You should arrange with the patent holders if you want to use this software within the United States of America for commercial purposes. Their patent claims cover a wide variety of the field of automatic uncertainty propagation. Therefore our extensions to their proposal may also be subject to the claims of that patent. In order to stop the spread of e-patents in Europe, please support us and sign the petition for Software Patent Free Europe (see link shown below).
There exists an alternative package for Python issued by the patent holders that allows the automatic propagation of uncertainty. Unfortunately this package does not provide any support for physical quantities and units. This package does also not integrate the standard numpy module and is therefore less flexible than our package.

Author:

Thomas Reidemeister

See also:

- **Installation Instructions** (p. 356).
- The Java Scientific Library (`http://www.jscience.org`)
- Java Specification Request - 275 (`http://www.jcp.org/jsr/detail/275.jsp`)
- "The "GUM Tree": A software design pattern for handling measurement uncertainty"; B. D. Hall; Industrial Research Report 1291; Measurements Standards Laboratory New Zealand (2003).
- "byGUM: A Python software package for calculating measurement uncertainty"; B. D. Hall; Industrial Research Report 1305; Measurements Standards Laboratory New Zealand (2005).

- Petition for a Software Patent Free Europe (http://www.noepatents.org/).
- United States Patent and Trademark Office (http://www.uspto.gov/patft/)
- PEP-239 - Adding a Rational Type to Python (http://www.python.org/dev/peps/pep-0239/)

2 SCUQ Module Documentation

2.1 The Arithmetic Module

2.1.1 Detailed Description

This module contains several functions, classes, and constants that are used for numeric computations in the other modules of this library.

Author:

Thomas Reidemeister

Classes

- class **RationalNumber**

  This class provides support for rational numbers.

Functions

- def **complex_to_matrix**

  This function converts a complex number to a column vector.

- def **gcd**

  Calculate the greatest common divisor.

- def **rational**

  This function provides an interface for rational numbers creation, as suggested in PEP 239.

Variables

- string **INFINITY** = "inf"

  Global constant for infinity that is used in combination with the degrees of freedom evaluation.

2.1.2 Function Documentation

2.1.2.1 def scuq::arithmetic::complex_to_matrix (*c*)

This function converts a complex number to a column vector.

Parameters:

c A complex number (a,b).

Returns:

An instance of numpy.matrix.

2.1.2.2 def scuq::arithmetic::gcd (*m*, *n*)

Calculate the greatest common divisor.

Parameters:

n First integer value (greater or equal to zero).

m Second value (greater or equal to zero).

Returns:

The greatest common divisor of the inputs.

2.1.2.3 def scuq::arithmetic::rational (*n*, *d*)

This function provides an interface for rational numbers creation, as suggested in PEP 239.

Parameters:

d The denominator (must be an interger type).

n The nominator (must be an integer type).

Returns:

An instance of **RationalNumber** (p. 210)

2.1.3 Variable Documentation

2.1.3.1 string INFINITY = "inf" [static]

Global constant for infinity that is used in combination with the degrees of freedom evaluation.

2.2 The Complex Uncertainty Module

2.2.1 Detailed Description

This module contains classes to model complex uncertain values.

Attention:

You should either use the module ucomponents or this module. Do not use both modules at once!

Author:

`Thomas Reidemeister`

Classes

- class **Abs**
  This class models taking the absolute value of a complex function.
- class **Add**
  This class models adding two complex values.
- class **ArcCos**
  This class models the inverse cosine function.
- class **ArcCosh**
  This class models the inverse hyperbolic cosine function.
- class **ArcSin**
  This class models the inverse sine function.
- class **ArcSinh**
  This class models the inverse hyperbolic sine function.
- class **ArcTan**
  This class models the inverse tangent function.
- class **ArcTan2**
  This class models two-argument inverse tangent.
- class **ArcTanh**
  This class models the inverse hyperbolic tangent function.
- class **CBinaryOperation**
  This abstract class models a binary operation.
- class **Conjugate**

This class models taking the negative of a complex value.

- class **Context**
  This class provides a context for complex-valued uncertainty evaluations. It manages the correlation coefficients and is able to evaluate the effective degrees of freedom.
- class **Cos**
  This class models the cosine function.
- class **Cosh**
  This class models the hyperbolic cosine function.
- class **CUnaryOperation**
  This abstract class models an unary operation.
- class **CUncertainComponent**
  This is the abstract super class of all complex valued uncertain components. Despite defining the interface for complex valued uncertain components, it also provides a set of factory methods that act as an interface for `numpy`.
- class **CUncertainInput**
  This class models a complex-valued input of a function.
- class **Div**
  This class models dividing two complex values.
- class **Exp**
  This class models the exponential function e^x. x denotes the sibling of this instance.
- class **Inv**
  This class models inverting complex values. Let an instance of this class model the complex value x then this class models $\frac{1}{x}$.
- class **Log**
  This class models logarithms having a real base. However, the base cannot be uncertain.
- class **Mul**
  This class models multiplying two complex values.
- class **Neg**
  This class models taking the negative of a complex value.
- class **Pow**
  This class models complex powers.
- class **Sin**

This class models the sine function.

- class **Sinh**
  This class models the hyperbolic sine function.
- class **Sqrt**
  This class models taking the square root of an uncertain component.
- class **Sub**
  This class models taking the difference of two complex values.
- class **Tan**
  This class models the tangent function.
- class **Tanh**
  This class models the hyperbolic tangent function.

Functions

- def **complex_to_matrix**
  This function transforms a complex value into a matrix.

2.2.2 Function Documentation

2.2.2.1 def scuq::cucomponents::complex_to_matrix (*value*)

This function transforms a complex value into a matrix.

Parameters:

value The complex value.

Returns:

A 2x2-matrix containing the value.

2.3 The Operators Module

2.3.1 Detailed Description

This module contains the classes necessary to define, handle, and use operators on units.

Author:

`Thomas Reidemeister`

Classes

- class **__ExpOperator__**
  This class provides an Interface for exponential operators. It is used as helper for the ***LogOperator*** (p. 156)*.*
- class **AddOperator**
  This class provides an Interface for offset operators.
- class **CompoundOperator**
  Compound Operator.
- class **Identity**
  This class provides an Interface for the identity Operator.
- class **LogOperator**
  This class provides an interface for logarithmic operators.
- class **MultiplyOperator**
  This class provides an Interface for factor operators.
- class **UnitOperator**
  Basic abstract Operator to use on units.

Variables

- tuple **IDENTITY** = Identity()

2.3.2 Variable Documentation

2.3.2.1 tuple IDENTITY = Identity() `[static]`

Global **Identity** (p. 148) Operator.

Since there is only one **Identity** (p. 148), it is defined global here.

2.4 The Exceptions Module

2.4.1 Detailed Description

This module contains the classes to model, handle, and use special qexceptions that may occur while using of units and quantities.

Author:

`Thomas Reidemeister`

Classes

- class **ConversionException**
  General exception that is raised whenever a unit conversion fails.
- class **NotDimensionlessException**
  Exception that is raised whenever a a unit is not dimensionless where it has to be.
- class **QuantitiesException**
  General class for qexceptions of this module.
- class **UnitExistsException**
  Exception that is raised when a dimension, base unit, or alternate unit of the same type has already been created.
- class **UnknownUnitException**
  An exception that is raised whenever an unexpected unit was used.

2.5 The Quantities Module

2.5.1 Detailed Description

This module contains the classes to model, handle, and use physical quantities. Because of Pythons nature of weak typing, this implementation strongly differs from the jsr-275. In our interpretation a **Quantity** (p. 184) is a tuple of a numeric value and a unit. Therefore we provide a class **Quantity** (p. 184) that emulates a numeric type and checks explicitly for consistency for each operation. Thus, this module is able to provide at least runtime checking for physical dimensions. For quantities you may choose between strict and non-strict unit checks. Strict type checking means that this instance raises an error whenever unequal units are compared. If strict checking is disabled this instance tries to convert among the units. An error will only be raised, if the units are not compatible (i.e. describe a different physical dimension). For example if you want to add a quantity measured in feet to a quantity measured in meters. If Strict type checking is enabled an error is raised. Otherwise the quantity measured in feet will be transformed to meters before being added. Strict type checking is enabled by default. In this class we use comparable for the case that the units can be converted to each other and no strict checking is used or for strict type checking and equal units.

See also:

Quantity (p. 184)

Author:

`Thomas Reidemeister`

See also:

Quantity.set_strict (p. 210)
Quantity.is_strict (p. 210)

Classes

- class **Quantity**
  Base class that provides an interface to model quantities.

Functions

- def **is_strict**
  *An abbreviation for **Quantity.is_strict** (p. 210).*
- def **set_strict**
  *An abbreviation for **Quantity.set_strict** (p. 210).*

2.5.2 Function Documentation

2.5.2.1 def scuq::quantities::is_strict ()

An abbreviation for **Quantity.is_strict** (p. 210).

2.5.2.2 def scuq::quantities::set_strict (*bValue* = `True`)

An abbreviation for **Quantity.set_strict** (p. 210).

Parameters:

bValue

2.6 The Standard System of Units Module

2.6.1 Detailed Description

This module contains the predefined SI units. It models SI base units and SI alternate units. The alternate units have been formed as product other alternate SI units where possible as described in NIST 330.

See also:

"The International System of Units"; Barry N. Taylor; NIST 330 (2001)

Author:

`Thomas Reidemeister`

Classes

- class **SIModel**
  The interface for a physical model for SI units.

Variables

- tuple **__model** = SIModel()
- tuple **AMPERE** = units.BaseUnit("A")
  Unit instance to model the BaseUnit Ampere.
- tuple **BECQUEREL** = units.AlternateUnit("Bq", ∼SECOND)
  Unit instance to model the SI unit Becquerel.
- tuple **CANDELA** = units.BaseUnit("cd")
  Unit instance to model the BaseUnit Candela.
- float **CELSIUS** = 273.15
  Unit instance to model the SI unit degree Celsius.
- tuple **COULOMB** = units.AlternateUnit("C", AMPERE ∗ SECOND)
  Unit instance to model the SI unit Coulomb.
- tuple **FARAD** = units.AlternateUnit("F", COULOMB / VOLT)
  Unit instance to model the SI unit Farad.
- tuple **GRAY** = units.AlternateUnit("Gy", JOULE/KILOGRAM)
  Unit instance to model the SI unit Gray.
- tuple **HENRY** = units.AlternateUnit("H", WEBER / AMPERE)
  Unit instance to model the SI unit Henry.
- tuple **HERTZ** = units.AlternateUnit("Hz", ∼SECOND)
  Unit instance to model the SI unit Herz.
- tuple **JOULE** = units.AlternateUnit("J", NEWTON ∗ METER)
  Unit instance to model the SI unit Joule.
- tuple **KATAL** = units.AlternateUnit("kat", MOLE/SECOND)
  Unit instance to model the SI unit Katal.
- tuple **KELVIN** = units.BaseUnit("K")
  Unit instance to model the BaseUnit Kelvin.
- tuple **KILOGRAM** = units.BaseUnit("kg")
  Unit instance to model the BaseUnit Kilogram.
- tuple **LUMEN** = units.AlternateUnit("lm", CANDELA∗STERADIAN)
  Unit instance to model the SI unit Lumen.
- tuple **LUX** = units.AlternateUnit("lx", LUMEN/(METER∗METER))

Unit instance to model the SI unit Lux.

- tuple **METER** = units.BaseUnit("m")
  Unit instance to model the BaseUnit Meter.
- tuple **MOLE** = units.BaseUnit("mol")
  Unit instance to model the BaseUnit Mol.
- tuple **NEWTON** = units.AlternateUnit("N", KILOGRAM ∗ METER/(SECOND ∗∗ 2))
  Unit instance to model the SI unit Newton.
- tuple **OHM**
- tuple **PASCAL** = units.AlternateUnit("Pa", NEWTON / (METER ∗∗ 2))
  Unit instance to model the SI unit Pascal.
- tuple **RADIAN** = units.AlternateUnit("rad", units.ONE)
- tuple **SECOND** = units.BaseUnit("s")
  Unit instance to model the BaseUnit Second.
- tuple **SIEMENS** = units.AlternateUnit("S", AMPERE / VOLT)
  Unit instance to model the SI unit Siemens.
- tuple **SIVERT** = units.AlternateUnit("Sv", JOULE/KILOGRAM)
  Unit instance to model the SI unit Sivert.
- tuple **STERADIAN** = units.AlternateUnit("sr", units.ONE)
- tuple **TESLA** = units.AlternateUnit("T", WEBER / (METER∗∗2))
  Unit instance to model the SI unit Tesla.
- tuple **VOLT** = units.AlternateUnit("V", WATT / AMPERE)
  Unit instance to model the SI unit Volt.
- tuple **WATT** = units.AlternateUnit("W", JOULE / SECOND)
  Unit instance to model the SI unit Watt.
- tuple **WEBER** = units.AlternateUnit("Wb", VOLT ∗ SECOND)
  Unit instance to model the SI unit Weber.

2.6.2 Variable Documentation

2.6.2.1 tuple __model = SIModel() [static]

2.6.2.2 tuple AMPERE = units.BaseUnit("A") [static]

Unit instance to model the BaseUnit Ampere.

2.6.2.3 tuple BECQUEREL = units.AlternateUnit("Bq", ∼SECOND) [static]

Unit instance to model the SI unit Becquerel.

2.6.2.4 tuple CANDELA = units.BaseUnit("cd") [static]

Unit instance to model the BaseUnit Candela.

2.6.2.5 float CELSIUS = 273.15 [static]

Unit instance to model the SI unit degree Celsius.

2.6.2.6 tuple COULOMB = units.AlternateUnit("C", AMPERE ∗ SECOND) [static]

Unit instance to model the SI unit Coulomb.

2.6.2.7 tuple FARAD = units.AlternateUnit("F", COULOMB / VOLT) [static]

Unit instance to model the SI unit Farad.

2.6.2.8 tuple GRAY = units.AlternateUnit("Gy", JOULE/KILOGRAM) [static]

Unit instance to model the SI unit Gray.

2.6.2.9 tuple HENRY = units.AlternateUnit("H", WEBER / AMPERE) [static]

Unit instance to model the SI unit Henry.

2.6.2.10 tuple HERTZ = units.AlternateUnit("Hz", ∼SECOND) [static]

Unit instance to model the SI unit Herz.

2.6.2.11 tuple JOULE = units.AlternateUnit("J", NEWTON ∗ METER) [static]

Unit instance to model the SI unit Joule.

2.6.2.12 tuple KATAL = units.AlternateUnit("kat", MOLE/SECOND) [static]

Unit instance to model the SI unit Katal.

2.6.2.13 tuple KELVIN = units.BaseUnit("K") [static]

Unit instance to model the BaseUnit Kelvin.

2.6.2.14 tuple KILOGRAM = units.BaseUnit("kg") [static]

Unit instance to model the BaseUnit Kilogram.

2.6.2.15 tuple LUMEN = units.AlternateUnit("lm", CANDELA∗STERADIAN) [static]

Unit instance to model the SI unit Lumen.

2.6.2.16 tuple LUX = units.AlternateUnit("lx", LUMEN/(METER∗METER)) [static]

Unit instance to model the SI unit Lux.

2.6.2.17 tuple METER = units.BaseUnit("m") [static]

Unit instance to model the BaseUnit Meter.

2.6.2.18 tuple MOLE = units.BaseUnit("mol") [static]

Unit instance to model the BaseUnit Mol.

2.6.2.19 tuple NEWTON = units.AlternateUnit("N", KILOGRAM ∗ METER/(SECOND ∗∗ 2)) [static]

Unit instance to model the SI unit Newton.

2.6.2.20 tuple OHM [static]

Initial value:

```
units.AlternateUnit( ( u"\u03A9" ).encode( "UTF-8" ),
                                    VOLT / AMPERE )
```

Unit instance to model the SI unit Ohm.

Note:

The UTF-8 encoded string stands for Ω.

2.6.2.21 tuple PASCAL = units.AlternateUnit("Pa", NEWTON / (METER ∗∗ 2)) [static]

Unit instance to model the SI unit Pascal.

2.6.2.22 tuple RADIAN = units.AlternateUnit("rad", units.ONE) [static]

Unit instance to model the SI unit Radian.

Attention:

Because this library keeps only the canonical form of the product of base units the Radian is compatible to the neutral 1. Therefore its system unit is modelled as "1" not as $\frac{m}{m}$.

2.6.2.23 tuple SECOND = units.BaseUnit("s") [static]

Unit instance to model the BaseUnit Second.

2.6.2.24 tuple SIEMENS = units.AlternateUnit("S", AMPERE / VOLT) [static]

Unit instance to model the SI unit Siemens.

2.6.2.25 tuple SIVERT = units.AlternateUnit("Sv", JOULE/KILOGRAM) [static]

Unit instance to model the SI unit Sivert.

2.6.2.26 tuple STERADIAN = units.AlternateUnit("sr", units.ONE) [static]

Unit instance to model the SI base unit Steradian.

Attention:

Because this library keeps only the canonical form of the product of base units the Radian is compatible to the neutral 1. Therefore its system unit is modelled as "1" not as $\frac{m^2}{m^2}$.

2.6.2.27 tuple TESLA = units.AlternateUnit("T", WEBER / (METER**2)) [static]

Unit instance to model the SI unit Tesla.

2.6.2.28 tuple VOLT = units.AlternateUnit("V", WATT / AMPERE) [static]

Unit instance to model the SI unit Volt.

2.6.2.29 tuple WATT = units.AlternateUnit("W", JOULE / SECOND) [static]

Unit instance to model the SI unit Watt.

2.6.2.30 tuple WEBER = units.AlternateUnit("Wb", VOLT * SECOND) [static]

Unit instance to model the SI unit Weber.

2.7 Testcases to verify the Quantitites library.

2.7.1 Detailed Description

This module contains a variety of test cases that verify this library.

Author:

Thomas Reidemeister

Classes

- class **TestArithmetic**
  This class provides the tests to verify the rational number module.
- class **TestComplexUncertaintyComponents**
  This class provides test-cases for the Module cucomponents.
- class **TestGUMTree**
  These classes test the function of the global elements of the GUM-tree, namely the Context class.
- class **TestOperators**
  Test the unit conversion operators.
- class **TestQuantity**
  This class provides the test cases for the quantities.
- class **TestSIUnits**
  SI Testing class. This class tests the definition and semantics of the SI units.
- class **TestUncertaintyComponents**
  This class provides tests for the ucomponents module.

Functions

- def **test_serialization**
  A general test for serialization of instances.

Variables

- tuple **suite** = unittest.TestSuite()

2.7.2 Function Documentation

2.7.2.1 def scuq::testcases::test_serialization (*instance*, *copy*, *sanityInstance*, *type*, *bCopy* = `True`)

A general test for serialization of instances.

Attention:

This test is only based on the __eq__ method of the instance. It is assumed that this method is working correctly.

Parameters:

instance The instance to serialize.

copy A copy of the instance (having other object reference).

sanityInstance An instance that is not equal to the instance.

type The type of the instance and the copy.

bCopy Use the above parameter copy (True,default), or apply only a weak check using no copy.

2.7.3 Variable Documentation

2.7.3.1 tuple suite = unittest.TestSuite() `[static]`

2.8 The Uncertainty Module

2.8.1 Detailed Description

This module contains classes to model uncertain values.

Author:

`Thomas Reidemeister`

Classes

- class **Abs**

  This class models the GUM-tree-nodes that take the absolute value of a silbling.

- class **Add**

  This class models GUM-tree nodes that add two silblings.

- class **ArcCos**

  This class models the GUM-tree-nodes that take the Arcus Cosine of a silbling.

- class **ArcCosh**

  This class models the GUM-tree-nodes that take the inverse Hyperbolic Cosine.

- class **ArcSin**

  This class models the GUM-tree-nodes that take the Arc Sine of a silbling.

- class **ArcSinh**

  This class models the GUM-tree-nodes that take the inverse Hyperbolic Sine of a silbling.

- class **ArcTan**

  This class models the GUM-tree-nodes that take the Arcus Tangent of a silbling.

- class **ArcTan2**

  This class models the inverse two-argument tangent.

- class **ArcTanh**

  This class models the GUM-tree-nodes that take the inverse Hyperbolic Tangent of a silbling.

- class **BinaryOperation**

  The abstract base class for modelling binary operations. This class provides the abstract interface for GUM-tree-nodes that have two silblings.

- class **Context**

  This class provides the context for an uncertainty evaluation. It maintains the correlation between the inputs and can be used to evaluate the combined standard uncertainty, as shown below. Let your model be $y = f(x_1, x_2, \ldots, x_N)$*, then* $u_c^2(y) = \sum_{i=1}^{N} \left(\frac{\delta f}{\delta x_i}\right)^2 u^2(x_i) + 2\sum_{i=1}^{N}\sum_{j=i+1}^{N} \frac{\delta f}{\delta x_i}\frac{\delta f}{\delta x_j} u(x_i, x_j)$.

- class **Cos**

  This class models the GUM-tree-nodes that take the Cosine of a silbling.

- class **Cosh**

  This class models the GUM-tree-nodes that take the Hyperbolic Cosine of a silbling.

- class **Div**

  This class models GUM-tree nodes that divide two silblings.

- class **Exp**

  This class models the GUM-tree-nodes that take the exponential of a silbling.

- class **Log**

  This class models the GUM-tree-nodes that take the Natural Logarithm of a silbling.

- class **Mul**

  This class models GUM-tree nodes that multiply two silblings.

- class **Neg**

  This class models the unary negation as GUM-tree-element.

- class **Pow**

  This class models GUM-tree nodes that raise the left silbling to the power of the right one.

- class **Sin**

  This class models the GUM-tree-nodes that take the Sine of a silbling.

- class **Sinh**

  This class models the GUM-tree-nodes that take the Hyperbolic Sine of a silbling.

- class **Sqrt**

  This class models the GUM-tree-nodes that take the square root of a silbling.

- class **Sub**

  This class models GUM-tree nodes that take the difference of the two silblings.

- class **Tan**

  This class models the GUM-tree-nodes that take the Tangent of a silbling.

- class **Tanh**

  This class models the GUM-tree-nodes that take the Hyperbolic Tangent of a silbling.

- class **UnaryOperation**

  The abstract base class for modelling unary operations. This class provides the abstract interface for GUM-tree-nodes that have one silbling.

- class **UncertainComponent**

  This is the abstract base class to model components of uncertainty as described in by "The GUM Tree".

- class **UncertainInput**

  This class provides the model for uncertain inputs, that are referred to as "Leafs" in "The GUM tree".

Functions

- def **clearDuplicates**

  Remove identical elements from a list.

2.8.2 Function Documentation

2.8.2.1 def scuq::ucomponents::clearDuplicates (*alist*)

Remove identical elements from a list.

Parameters:

alist A list that may contain identical elements.

Returns:

A list that only contains unique elements.

2.9 Units Module

2.9.1 Detailed Description

This module contains the classes necessary to define, handle, and work with physical units and dimensions.

Author:

`Thomas Reidemeister`

Classes

- class **__ProductElement__**

  A helper class for ***ProductUnit*** *(*p. 176*) classes. This class helps to maintain the factors of a product unit.*

- class **AlternateUnit**

  This class provides an interface for units that describe the same dimension as another unit, but need to be distinguished from it by another symbol (e.g. to abbreviate them, or to distinguish their purpose).

- class **BaseUnit**

  This class provides the interface to define and use base units.

- class **CompoundUnit**

  This class provides an interface for describing compound units. The units forming a compound unit have to describe the same physical dimension. For example time $[hour : min : second]$.

- class **DerivedUnit**

  This class provides an abstract interface for all units that have been transformed from other units.

- class **Dimension**

  This class provides an interface to model physical dimensions.

- class **PhysicalModel**

  This class models the abstract interface for physical models.

- class **ProductUnit**

  The unit is a combined unit of the product of the powers of units.

- class **TransformedUnit**

  This class provides an interface for a unit that has been derived from a unit using an operator.

- class **Unit**

  An abstract class to model physical units.

- class **UnitsManager**

  This manages the alternate and base units as well as the physical dimensions.

Functions

- def **get_default_model**

  Get the physical model currently in use. This function returns None, if no model is currently in use.

- def **set_default_model**

  Set the default physical model to use.

Variables

- tuple **__char** = __unicode.encode("UTF-8")
- string **__unicode** = u"\u03b8"
- tuple **__UNITS_MANAGER__** = UnitsManager()

  Global units Manager that keeps track of the units and dimensions created.

- tuple **CURRENT** = Dimension("I")

  Predefined global dimension for the Electric Current.

- tuple **LENGTH** = Dimension("L")

  Predefined global dimension for the Length.

- tuple **LUMINOUS_INTENSITY** = Dimension("Li")

  Predefined global dimension for Luminous Intensity.

- tuple **MASS** = Dimension("M")

  Predefined global dimension for the Mass.

- tuple **NONE** = Dimension(ONE)

  Predefined global dimension for a dimensionless quantity.

- tuple **ONE** = ProductUnit()

  Dimensionless unit ONE.

- tuple **SUBSTANCE** = Dimension("n")

  Predefined global dimension for the Amount of Substance.

- tuple **TEMPERATURE** = Dimension(__char)

  Predefined global dimension for the Temperature.

- tuple **TIME** = Dimension("t")

  Predefined global dimension for the Time.

2.9.2 Function Documentation

2.9.2.1 def scuq::units::get_default_model ()

Get the physical model currently in use. This function returns None, if no model is currently in use.

Returns:

The physical model that is currently in use.

See also:

PhysicalModel (p. 172).

2.9.2.2 def scuq::units::set_default_model (*physicalModel*)

Set the default physical model to use.

Parameters:

physicalModel The physical model to use.

See also:

PhysicalModel (p. 172).

2.9.3 Variable Documentation

2.9.3.1 tuple __char = __unicode.encode("UTF-8") [static]

2.9.3.2 string __unicode = u"\u03b8" [static]

2.9.3.3 tuple __UNITS_MANAGER__ = UnitsManager() [static]

Global units Manager that keeps track of the units and dimensions created.

2.9.3.4 tuple CURRENT = Dimension("I") [static]

Predefined global dimension for the Electric Current.

2.9.3.5 tuple LENGTH = Dimension("L") [static]

Predefined global dimension for the Length.

2.9.3.6 tuple LUMINOUS_INTENSITY = Dimension("Li") [static]

Predefined global dimension for Luminous Intensity.

2.9.3.7 tuple MASS = Dimension("M") [static]

Predefined global dimension for the Mass.

2.9.3.8 tuple NONE = Dimension(ONE) [static]

Predefined global dimension for a dimensionless quantity.

2.9.3.9 tuple ONE = ProductUnit() [static]

Dimensionless unit ONE.

2.9.3.10 tuple SUBSTANCE = Dimension("n") [static]

Predefined global dimension for the Amount of Substance.

2.9.3.11 tuple TEMPERATURE = Dimension(__char) [static]

Predefined global dimension for the Temperature.

2.9.3.12 tuple TIME = Dimension("t") [static]

Predefined global dimension for the Time.

3 SCUQ Namespace Documentation

3.1 scuq Namespace Reference

3.1.1 Detailed Description

The namespace containing this library.

Namespaces

- namespace **__init__**

  This namespace does only contain variables for global initialization.

- namespace **arithmetic**

  This namespace contains several functions and classes that are used for numeric computations within this class library.

- namespace **cucomponents**

  This namespace contains the classes to evaluate the uncertainty of complex-valued functions.

- namespace **operators**

  This namespace contains operators that are used for unit conversions.

- namespace **qexceptions**

  This namespace contains classes defining custom exceptions of this library.

- namespace **quantities**

  *This namespace contains the class **Quantity** (p. 184) that models physical quantities.*

- namespace **si**

  This namespace contains the SI-units.

- namespace **testcases**

  This namespace contains several test cases to validate and verify this class library.

- namespace **ucomponents**

  This namespace contains the classes to evaluate the uncertainty of scalar functions.

- namespace **units**

  This namespace contains the classes and constants to model units.

3.2 scuq::__init__ Namespace Reference

3.2.1 Detailed Description

This namespace does only contain variables for global initialization.

Variables

- list **__all__** = ["arithmetic", "units", "qexceptions", "si", "quantities", "operators", "ucomponents", "cucomponents"]

  The modules contained within the quantities package.

3.2.2 Variable Documentation

3.2.2.1 list __all__ = ["arithmetic", "units", "qexceptions", "si", "quantities", "operators", "ucomponents", "cucomponents"] [static]

The modules contained within the quantities package.

3.3 scuq::arithmetic Namespace Reference

3.3.1 Detailed Description

This namespace contains several functions and classes that are used for numeric computations within this class library.

Classes

- class **RationalNumber**

  This class provides support for rational numbers.

Functions

- def **complex_to_matrix**

  This function converts a complex number to a column vector.

- def **gcd**

  Calculate the greatest common divisor.

- def **rational**

  This function provides an interface for rational numbers creation, as suggested in PEP 239.

Variables

- string **INFINITY** = "inf"

  Global constant for infinity that is used in combination with the degrees of freedom evaluation.

3.4 scuq::cucomponents Namespace Reference

3.4.1 Detailed Description

This namespace contains the classes to evaluate the uncertainty of complex-valued functions.

Classes

- class **Abs**

  This class models taking the absolute value of a complex function.

- class **Add**

  This class models adding two complex values.

- class **ArcCos**

  This class models the inverse cosine function.

- class **ArcCosh**

  This class models the inverse hyperbolic cosine function.

- class **ArcSin**

  This class models the inverse sine function.

- class **ArcSinh**

  This class models the inverse hyperbolic sine function.

- class **ArcTan**

  This class models the inverse tangent function.

- class **ArcTan2**

  This class models two-argument inverse tangent.

- class **ArcTanh**

  This class models the inverse hyperbolic tangent function.

- class **CBinaryOperation**

  This abstract class models a binary operation.

- class **Conjugate**

  This class models taking the negative of a complex value.

- class **Context**

  This class provides a context for complex-valued uncertainty evaluations. It manages the correlation coefficients and is able to evaluate the effective degrees of freedom.

- class **Cos**

  This class models the cosine function.

- class **Cosh**

  This class models the hyperbolic cosine function.

- class **CUnaryOperation**

  This abstract class models an unary operation.

- class **CUncertainComponent**

  This is the abstract super class of all complex valued uncertain components. Despite defining the interface for complex valued uncertain components, it also provides a set of factory methods that act as an interface for `numpy`.

- class **CUncertainInput**

  This class models a complex-valued input of a function.

- class **Div**

  This class models dividing two complex values.

- class **Exp**

  This class models the exponential function e^x. x denotes the sibling of this instance.

- class **Inv**

  This class models inverting complex values. Let an instance of this class model the complex value x then this class models $\frac{1}{x}$.

- class **Log**

  This class models logarithms having a real base. However, the base cannot be uncertain.

- class **Mul**

  This class models multiplying two complex values.

- class **Neg**

  This class models taking the negative of a complex value.

- class **Pow**

  This class models complex powers.

- class **Sin**

  This class models the sine function.

- class **Sinh**

  This class models the hyperbolic sine function.

- class **Sqrt**

  This class models taking the square root of an uncertain component.

- class **Sub**

  This class models taking the difference of two complex values.

- class **Tan**

  This class models the tangent function.

- class **Tanh**

  This class models the hyperbolic tangent function.

Functions

- def **complex_to_matrix**

  This function transforms a complex value into a matrix.

3.5 scuq::operators Namespace Reference

3.5.1 Detailed Description

This namespace contains operators that are used for unit conversions.

Classes

- class **__ExpOperator__**

  This class provides an Interface for exponential operators. It is used as helper for the ***LogOperator*** *(p. 156).*

- class **AddOperator**

  This class provides an Interface for offset operators.

- class **CompoundOperator**

  Compound Operator.

- class **Identity**

  This class provides an Interface for the identity Operator.

- class **LogOperator**
  This class provides an interface for logarithmic operators.
- class **MultiplyOperator**
  This class provides an Interface for factor operators.
- class **UnitOperator**
  Basic abstract Operator to use on units.

Variables

- tuple **IDENTITY = Identity**()

3.6 scuq::qexceptions Namespace Reference

3.6.1 Detailed Description

This namespace contains classes defining custom exceptions of this library.

Classes

- class **ConversionException**
  General exception that is raised whenever a unit conversion fails.
- class **NotDimensionlessException**
  Exception that is raised whenever a a unit is not dimensionless where it has to be.
- class **QuantitiesException**
  General class for qexceptions of this module.
- class **UnitExistsException**
  Exception that is raised when a dimension, base unit, or alternate unit of the same type has already been created.
- class **UnknownUnitException**
  An exception that is raised whenever an unexpected unit was used.

3.7 scuq::quantities Namespace Reference

3.7.1 Detailed Description

This namespace contains the class **Quantity** (p. 184) that models physical quantities.

Classes

- class **Quantity**
  Base class that provides an interface to model quantities.

Functions

- def **is_strict**
  *An abbreviation for **Quantity.is_strict** (p. 210).*
- def **set_strict**
  *An abbreviation for **Quantity.set_strict** (p. 210).*

3.8 scuq::si Namespace Reference

3.8.1 Detailed Description

This namespace contains the SI-units.

Classes

- class **SIModel**
  The interface for a physical model for SI units.

Variables

- tuple **__model = SIModel**()
- tuple **AMPERE = units.BaseUnit**("A")
  Unit instance to model the BaseUnit Ampere.
- tuple **BECQUEREL = units.AlternateUnit**("Bq", ∼SECOND)
  Unit instance to model the SI unit Becquerel.
- tuple **CANDELA = units.BaseUnit**("cd")
  Unit instance to model the BaseUnit Candela.
- float **CELSIUS** = 273.15
  Unit instance to model the SI unit degree Celsius.
- tuple **COULOMB = units.AlternateUnit**("C", AMPERE ∗ **SECOND**)
  Unit instance to model the SI unit Coulomb.

- tuple **FARAD = units.AlternateUnit**("F", COULOMB **/ VOLT**)
  Unit instance to model the SI unit Farad.
- tuple **GRAY = units.AlternateUnit**("Gy", JOULE**/KILOGRAM**)
  Unit instance to model the SI unit Gray.
- tuple **HENRY = units.AlternateUnit**("H", WEBER **/ AMPERE**)
  Unit instance to model the SI unit Henry.
- tuple **HERTZ = units.AlternateUnit**("Hz", ∼SECOND)
  Unit instance to model the SI unit Herz.
- tuple **JOULE = units.AlternateUnit**("J", NEWTON ∗ **METER**)
  Unit instance to model the SI unit Joule.
- tuple **KATAL = units.AlternateUnit**("kat", MOLE**/SECOND**)
  Unit instance to model the SI unit Katal.
- tuple **KELVIN = units.BaseUnit**("K")
  Unit instance to model the BaseUnit Kelvin.
- tuple **KILOGRAM = units.BaseUnit**("kg")
  Unit instance to model the BaseUnit Kilogram.
- tuple **LUMEN = units.AlternateUnit**("lm", CANDELA∗**STERADIAN**)
  Unit instance to model the SI unit Lumen.
- tuple **LUX = units.AlternateUnit**("lx", LUMEN/(**METER**∗**METER**))
  Unit instance to model the SI unit Lux.
- tuple **METER = units.BaseUnit**("m")
  Unit instance to model the BaseUnit Meter.
- tuple **MOLE = units.BaseUnit**("mol")
  Unit instance to model the BaseUnit Mol.
- tuple **NEWTON = units.AlternateUnit**("N", KILOGRAM ∗ **METER/**(**SECOND** ∗∗ 2))
  Unit instance to model the SI unit Newton.
- tuple **OHM**
- tuple **PASCAL = units.AlternateUnit**("Pa", NEWTON / (**METER** ∗∗ 2))
  Unit instance to model the SI unit Pascal.
- tuple **RADIAN = units.AlternateUnit**("rad", units.ONE)
- tuple **SECOND = units.BaseUnit**("s")

Unit instance to model the BaseUnit Second.

- tuple **SIEMENS = units.AlternateUnit**("S", AMPERE **/ VOLT**)
  Unit instance to model the SI unit Siemens.
- tuple **SIVERT = units.AlternateUnit**("Sv", JOULE**/KILOGRAM**)
  Unit instance to model the SI unit Sivert.
- tuple **STERADIAN = units.AlternateUnit**("sr", units.ONE)
- tuple **TESLA = units.AlternateUnit**("T", WEBER **/** (**METER**∗∗2))
  Unit instance to model the SI unit Tesla.
- tuple **VOLT = units.AlternateUnit**("V", WATT **/ AMPERE**)
  Unit instance to model the SI unit Volt.
- tuple **WATT = units.AlternateUnit**("W", JOULE **/ SECOND**)
  Unit instance to model the SI unit Watt.
- tuple **WEBER = units.AlternateUnit**("Wb", VOLT ∗ **SECOND**)
  Unit instance to model the SI unit Weber.

3.9 scuq::testcases Namespace Reference

3.9.1 Detailed Description

This namespace contains several test cases to validate and verify this class library.

Classes

- class **TestArithmetic**
  This class provides the tests to verify the rational number module.
- class **TestComplexUncertaintyComponents**
  This class provides test-cases for the Module cucomponents.
- class **TestGUMTree**
  These classes test the function of the global elements of the GUM-tree, namely the Context class.
- class **TestOperators**
  Test the unit conversion operators.
- class **TestQuantity**
  This class provides the test cases for the quantities.

- class **TestSIUnits**

  SI Testing class. This class tests the definition and semantics of the SI units.

- class **TestUncertaintyComponents**

  This class provides tests for the ucomponents module.

Functions

- def **test_serialization**

  A general test for serialization of instances.

Variables

- tuple **suite** = unittest.TestSuite()

3.10 scuq::ucomponents Namespace Reference

3.10.1 Detailed Description

This namespace contains the classes to evaluate the uncertainty of scalar functions.

Classes

- class **Abs**

  This class models the GUM-tree-nodes that take the absolute value of a silbling.

- class **Add**

  This class models GUM-tree nodes that add two silblings.

- class **ArcCos**

  This class models the GUM-tree-nodes that take the Arcus Cosine of a silbling.

- class **ArcCosh**

  This class models the GUM-tree-nodes that take the inverse Hyperbolic Cosine.

- class **ArcSin**

  This class models the GUM-tree-nodes that take the Arc Sine of a silbling.

- class **ArcSinh**

  This class models the GUM-tree-nodes that take the inverse Hyperbolic Sine of a silbling.

- class **ArcTan**

This class models the GUM-tree-nodes that take the Arcus Tangent of a silbling.

- class **ArcTan2**

  This class models the inverse two-argument tangent.

- class **ArcTanh**

  This class models the GUM-tree-nodes that take the inverse Hyperbolic Tangent of a silbling.

- class **BinaryOperation**

  The abstract base class for modelling binary operations. This class provides the abstract interface for GUM-tree-nodes that have two silblings.

- class **Context**

  This class provides the context for an uncertainty evaluation. It maintains the correlation between the inputs and can be used to evaluate the combined standard uncertainty, as shown below. Let your model be $y = f(x_1, x_2, \ldots, x_N)$*, then* $u_c^2(y) = \sum_{i=1}^{N} \left(\frac{\delta f}{\delta x_i}\right)^2 u^2(x_i) + 2\sum_{i=1}^{N}\sum_{j=i+1}^{N} \frac{\delta f}{\delta x_i}\frac{\delta f}{\delta x_j} u(x_i, x_j)$.

- class **Cos**

  This class models the GUM-tree-nodes that take the Cosine of a silbling.

- class **Cosh**

  This class models the GUM-tree-nodes that take the Hyperbolic Cosine of a silbling.

- class **Div**

  This class models GUM-tree nodes that divide two silblings.

- class **Exp**

  This class models the GUM-tree-nodes that take the exponential of a silbling.

- class **Log**

  This class models the GUM-tree-nodes that take the Natural Logarithm of a silbling.

- class **Mul**

  This class models GUM-tree nodes that multiply two silblings.

- class **Neg**

  This class models the unary negation as GUM-tree-element.

- class **Pow**

  This class models GUM-tree nodes that raise the left silbling to the power of the right one.

- class **Sin**

  This class models the GUM-tree-nodes that take the Sine of a silbling.

- class **Sinh**

  This class models the GUM-tree-nodes that take the Hyperbolic Sine of a silbling.

- class **Sqrt**

  This class models the GUM-tree-nodes that take the square root of a silbling.

- class **Sub**

  This class models GUM-tree nodes that take the difference of the two silblings.

- class **Tan**

  This class models the GUM-tree-nodes that take the Tangent of a silbling.

- class **Tanh**

  This class models the GUM-tree-nodes that take the Hyperbolic Tangent of a silbling.

- class **UnaryOperation**

  The abstract base class for modelling unary operations. This class provides the abstract interface for GUM-tree-nodes that have one silbling.

- class **UncertainComponent**

  This is the abstract base class to model components of uncertainty as described in by "The GUM Tree".

- class **UncertainInput**

  This class provides the model for uncertain inputs, that are referred to as "Leafs" in "The GUM tree".

Functions

- def **clearDuplicates**

  Remove identical elements from a list.

3.11 scuq::units Namespace Reference

3.11.1 Detailed Description

This namespace contains the classes and constants to model units.

Classes

- class **__ProductElement__**

  *A helper class for **ProductUnit** (p. 176) classes. This class helps to maintain the factors of a product unit.*

- class **AlternateUnit**

  This class provides an interface for units that describe the same dimension as another unit, but need to be distinguished from it by another symbol (e.g. to abbreviate them, or to distinguish their purpose).

- class **BaseUnit**

  This class provides the interface to define and use base units.

- class **CompoundUnit**

  This class provides an interface for describing compound units. The units forming a compound unit have to describe the same physical dimension. For example time $[hour : min : second]$.

- class **DerivedUnit**

  This class provides an abstract interface for all units that have been transformed from other units.

- class **Dimension**

  This class provides an interface to model physical dimensions.

- class **PhysicalModel**

  This class models the abstract interface for physical models.

- class **ProductUnit**

  The unit is a combined unit of the product of the powers of units.

- class **TransformedUnit**

  This class provides an interface for a unit that has been derived from a unit using an operator.

- class **Unit**

  An abstract class to model physical units.

- class **UnitsManager**

  This manages the alternate and base units as well as the physical dimensions.

Functions

- def **get_default_model**

  Get the physical model currently in use. This function returns None, if no model is currently in use.

- def **set_default_model**

  Set the default physical model to use.

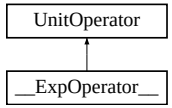
Variables

- tuple **__char** = __unicode.encode("UTF-8")
- string **__unicode** = u"\u03b8"
- tuple **__UNITS_MANAGER__** = **UnitsManager**()
  Global units Manager that keeps track of the units and dimensions created.
- tuple **CURRENT** = **Dimension**("I")
  Predefined global dimension for the Electric Current.
- tuple **LENGTH** = **Dimension**("L")
  Predefined global dimension for the Length.
- tuple **LUMINOUS_INTENSITY** = **Dimension**("Li")
  Predefined global dimension for Luminous Intensity.
- tuple **MASS** = **Dimension**("M")
  Predefined global dimension for the Mass.
- tuple **NONE** = **Dimension**(**ONE**)
  Predefined global dimension for a dimensionless quantity.
- tuple **ONE** = **ProductUnit**()
  Dimensionless unit ONE.
- tuple **SUBSTANCE** = **Dimension**("n")
  Predefined global dimension for the Amount of Substance.
- tuple **TEMPERATURE** = **Dimension**(**__char**)
  Predefined global dimension for the Temperature.
- tuple **TIME** = **Dimension**("t")
  Predefined global dimension for the Time.

4 SCUQ Class Documentation

4.1 __ExpOperator__ Class Reference

Inheritance diagram for __ExpOperator__::



4.1.1 Detailed Description

This class provides an Interface for exponential operators. It is used as helper for the **LogOperator** (p. 156).

Note:

Instances of this class can be serialized using pickle.

Public Member Functions

- def **__eq__**
  Test for equality.
- def **__getstate__**
  Serialization using pickle.
- def **__init__**
  Default constructor.
- def **__invert__**
  Invert this operation.
- def **__setstate__**
  Deserialization using pickle.
- def **__str__**
  Represent this operation by a string.
- def **convert**
  Convert a value.
- def **get_exponent**
  Get the base of logarithm.
- def **is_linear**
  Check if the operator is linear.

Private Attributes

- **__exponent__**
- **__logExponent__**

4.1.2 Member Function Documentation

4.1.2.1 def __eq__ (*self*, *other*)

Test for equality.

Parameters:

self

other Another **UnitOperator** (p. 326).

Reimplemented from **UnitOperator** (p. 327).

4.1.2.2 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **UnitOperator** (p. 327).

4.1.2.3 def __init__ (*self*, *exponent*)

Default constructor.

Initializes the operator and assigns the base to the current operator.

Parameters:

self

exponent the exponent.

4.1.2.4 def __invert__ (*self*)

Invert this operation.

This method returns the inverse operation of the current operation.

Parameters:

self

Returns:

The inverse operation of the current operation.

Reimplemented from **UnitOperator** (p. 327).

4.1.2.5 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

Reimplemented from **UnitOperator** (p. 328).

4.1.2.6 def __str__ (*self*)

Represent this operation by a string.

Parameters:

self

Returns:

A string describing this operation.

Reimplemented from **UnitOperator** (p. 328).

4.1.2.7 def convert (*self*, *value*)

Convert a value.

This method performs raises the current value to the exponent.

Parameters:

self

value The value to convert.

Returns:

The converted value

Reimplemented from **UnitOperator** (p. 329).

4.1.2.8 def get_exponent (*self*)

Get the base of logarithm.

This method returns the exponent.

Parameters:

self

Returns:

The base of the logarithm

4.1.2.9 def is_linear (*self*)

Check if the operator is linear.

This operator is not linear.

Parameters:

self

Returns:

False; this operator is not linear.

Reimplemented from **UnitOperator** (p. 329).

4.1.3 Member Data Documentation

4.1.3.1 __exponent__ [private]

4.1.3.2 __logExponent__ [private]

4.2 __ProductElement__ Class Reference

4.2.1 Detailed Description

A helper class for **ProductUnit** (p. 176) classes. This class helps to maintain the factors of a product unit.

Note:

Instances of this class can be serialized using pickle.

Public Member Functions

- def **__eq__**
  This method checks two factors for equality. Two factors are equal if they have the same units, powers, and roots.
- def **__getstate__**
  Serialization using pickle.
- def **__init__**
  Default constructor.
- def **__setstate__**
  Deserialization using pickle.

- def **__str__**
  Print this factor. This function returns a string of the form `unit^(pow/root)`.
- def **clone**
  Return a new instance of this factor.
- def **get_pow**
  Get the power of this factor.
- def **get_root**
  Get the root of this factor.
- def **get_unit**
  Get the unit of this factor.
- def **normalize**
  Transform the current factor into its canonical form.
- def **set_pow**
  This method changes the power.
- def **set_root**
  This method changes the root.

Static Private Attributes

- **__pow__** = None
  The power of the current factor.
- **__root__** = None
  The root of the current factor.
- **__unit__** = None
  The unit of the current factor.

4.2.2 Member Function Documentation

4.2.2.1 def __eq__ (*self*, *other*)

This method checks two factors for equality. Two factors are equal if they have the same units, powers, and roots.

Parameters:

self

other Another instance of a factor.

Returns:

True, if the factors are equal.

4.2.2.2 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

4.2.2.3 def __init__ (*self*, *unit*, *pow*, *root*)

Default constructor.

Parameters:

self
unit The unit of the factor to create.
pow The power assigned to this factor.
root The root assigned to this factor.

4.2.2.4 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self
state The state of the object.

4.2.2.5 def __str__ (*self*)

Print this factor. This function returns a string of the form `unit^(pow/root)`.

Parameters:

self

Returns:

A string describing this factor.

4.2.2.6 def clone (*self*)

Return a new instance of this factor.

Parameters:

self

Returns:

A new instance of this factor.

4.2.2.7 def get_pow (*self*)

Get the power of this factor.

Parameters:

self

Returns:

The power of this factor.

4.2.2.8 def get_root (*self*)

Get the root of this factor.

Parameters:

self

Returns:

The root of this factor.

4.2.2.9 def get_unit (*self*)

Get the unit of this factor.

Parameters:

self

Returns:

The unit of this factor.

4.2.2.10 def normalize (*self*)

Transform the current factor into its canonical form.

Parameters:

self

4.2.2.11 def set_pow (*self*, *value*)

This method changes the power.

Parameters:

self

value An interget to be used as new power.

4.2.2.12 def set_root (*self*, *value*)

This method changes the root.

Parameters:

self

value An interger to be used as new root.

4.2.3 Member Data Documentation

4.2.3.1 __pow__ = None [static, private]

The power of the current factor.

4.2.3.2 __root__ = None [static, private]

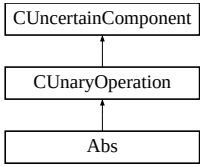
The root of the current factor.

4.2.3.3 __unit__ = None [static, private]

The unit of the current factor.

4.3 Abs Class Reference

Inheritance diagram for Abs::



4.3.1 Detailed Description

This class models taking the absolute value of a complex function.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.3.2 Member Function Documentation

4.3.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.3.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

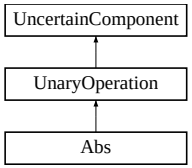
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.4 Abs Class Reference

Inheritance diagram for Abs::



4.4.1 Detailed Description

This class models the GUM-tree-nodes that take the absolute value of a silbling.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = |x|$ then the resulting uncertainty is $u(y) = |u(x)|$.
- def **get_value**
  Returns the exponential of the silbling.

4.4.2 Member Function Documentation

4.4.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self

right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.4.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.4.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = |x|$ then the resulting uncertainty is $u(y) = |u(x)|$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.4.2.4 def get_value (*self*)

Returns the exponential of the silbling.

Parameters:

self

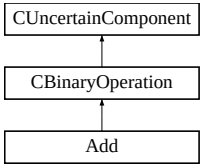
Returns:

A numeric value, representing the absolute value of the silbling.

Reimplemented from **UncertainComponent** (p. 305).

4.5 Add Class Reference

Inheritance diagram for Add::



4.5.1 Detailed Description

This class models adding two complex values.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.5.2 Member Function Documentation

4.5.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self
x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.5.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

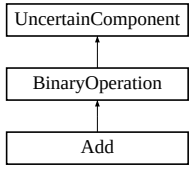
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.6 Add Class Reference

Inheritance diagram for Add::



4.6.1 Detailed Description

This class models GUM-tree nodes that add two silblings.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = x_1 + x_2$ then the resulting uncertainty is $u(y) = u(x_1) + u(x_2)$.
- def **get_value**
  Returns the sum of the silblings assigned.

4.6.2 Member Function Documentation

4.6.2.1 def __init__ (*self*, *left*, *right*)

Default constructor.

Parameters:

self
left Left silbling of this instance.
right Right silbling of this instance.

Reimplemented from **BinaryOperation** (p. 89).

4.6.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **BinaryOperation** (p. 89).

4.6.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = x_1 + x_2$ then the resulting uncertainty is $u(y) = u(x_1) + u(x_2)$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.6.2.4 def get_value (*self*)

Returns the sum of the silblings assigned.

Parameters:

self

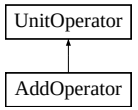
Returns:

A numeric value, representing the sum of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.7 AddOperator Class Reference

Inheritance diagram for AddOperator::



4.7.1 Detailed Description

This class provides an Interface for offset operators.

This class adds a constant value to an existing Operator.

Note:

Instances of this class can be serialized using pickle.

Public Member Functions

- def **__eq__**
  Test for equality.
- def **__getstate__**
  Serialization using pickle.
- def **__init__**
  Default constructor.
- def **__invert__**
  Invert the current operation.
- def **__mul__**
  Perform the current operation on another operator.
- def **__setstate__**
  Deserialization using pickle.
- def **__str__**
  Represent this operation by a string.
- def **convert**
  Convert a value.
- def **get_offset**
  Get the offset.
- def **is_linear**
  Check if this operator is linear.

Private Member Functions

- def **__isNegative**

  Helper method to optimize comparsions.

Private Attributes

- **__offset__**

4.7.2 Member Function Documentation

4.7.2.1 def __eq__ (*self*, *other*)

Test for equality.

Parameters:

self

other Another **UnitOperator** (p. 326).

Reimplemented from **UnitOperator** (p. 327).

4.7.2.2 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **UnitOperator** (p. 327).

4.7.2.3 def __init__ (*self*, *offset*)

Default constructor.

Initializes the operator and assigns the offset to the current operator.

Parameters:

self

offset The offset of this operator.

4.7.2.4 def __invert__ (*self*)

Invert the current operation.

This method returns the inverse operation of the current operation.

Parameters:

self

Returns:

The inverse operation of this operation.

Reimplemented from **UnitOperator** (p. 327).

4.7.2.5 def __isNegative (*positvieOp*, *negativeOp*) [private]

Helper method to optimize comparsions.

Parameters:

negativeOp An **AddOperator** (p. 51).

positvieOp An **AddOperator** (p. 51).

Returns:

```
negativeOp.get_offset() == -positvieOp.get_offset()
```

4.7.2.6 def __mul__ (*self*, *otherOperator*)

Perform the current operation on another operator.

The current operation (adding an offset a) will be performed on another operator $f(x)$. So that the new Operator is $a + g(x)$. If the other Operator is an **AddOperator** (p. 51), the offset is updated.

Parameters:

self

otherOperator The other operator to concat.

Returns:

The resulting operator.

Reimplemented from **UnitOperator** (p. 328).

4.7.2.7 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

Reimplemented from **UnitOperator** (p. 328).

4.7.2.8 def __str__ (*self*)

Represent this operation by a string.

Parameters:

self

Returns:

A string describing this operation.

Reimplemented from **UnitOperator** (p. 328).

4.7.2.9 def convert (*self*, *value*)

Convert a value.

This method performs the addition of an offset on an absolute value.

Parameters:

self

value The value to convert.

Returns:

The converted value

Reimplemented from **UnitOperator** (p. 329).

4.7.2.10 def get_offset (*self*)

Get the offset.

This method returns the offset of this operator.

Parameters:

self

Returns:

The offset of this operator

4.7.2.11 def is_linear (*self*)

Check if this operator is linear.

This operator is not linear.

Parameters:

self

Returns:

False

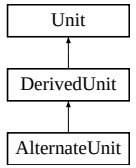
Reimplemented from **UnitOperator** (p. 329).

4.7.3 Member Data Documentation

4.7.3.1 __offset__ [private]

4.8 AlternateUnit Class Reference

Inheritance diagram for AlternateUnit::



4.8.1 Detailed Description

This class provides an interface for units that describe the same dimension as another unit, but need to be distinguished from it by another symbol (e.g. to abbreviate them, or to distinguish their purpose).

For examle $[N] := \left[\frac{kg \times m}{s^2}\right]$.

Note:

Instances of this class can be serialized using pickle.

Public Member Functions

- def **__eq__**
  Checks if two alternate units are equal.

- def **__getstate__**

  Serialization using pickle.

- def **__init__**

  Default constructor.

- def **__setstate__**

  Deserialization using pickle.

- def **__str__**

  *Print the current unit. This function is an alias for **AlternateUnit.get_symbol** (p. 59).*

- def **get_parent**

  Returns the parent unit of the this unit.

- def **get_symbol**

  Returns the symbol of this unit.

- def **get_system_unit**

  Returns the corresponding system unit. Since the parent unit is a system unit, this unit is supposed to be a system unit too. Therefore this function returns this instance.

- def **to_system_unit**

  *Get the operator to convert to the system unit. Since the parent unit is a system unit, this unit is supposed to be a system unit too. Therefore this function returns **operators.IDENTITY** (p. 8).*

Static Private Attributes

- **__parentUnit__** = None

  System unit that parents this unit.

- **__symbol__** = None

  Symbol for the alternate unit.

4.8.2 Member Function Documentation

4.8.2.1 def __eq__ (*self*, *other*)

Checks if two alternate units are equal.

Parameters:

self

other Another alternate unit to compare to.

Returns:

True, if the units are equal.

Reimplemented from **Unit** (p. 317).

4.8.2.2 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **Unit** (p. 317).

4.8.2.3 def __init__ (*self*, *symbol*, *parentUnit*)

Default constructor.

Parameters:

self

symbol Symbol of the alternate **Unit** (p. 314).

parentUnit Parent unit.

Exceptions:

UnitExistsException If a unit having the same symbol already exists.

TypeError If the parentUnit is not a SystemUnit.

4.8.2.4 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

Exceptions:

UnknownUnitException If the unit to be unpickled is not contained in the global repository __UNITS_MANAGER__.

See also:

UnitsManager (p. 329)
__UNITS_MANAGER__ (p. 23)

Reimplemented from **Unit** (p. 320).

4.8.2.5 def __str__ (*self*)

Print the current unit. This function is an alias for **AlternateUnit.get_symbol** (p. 59).

Parameters:

self

Returns:

A string describing this unit.

See also:

AlternateUnit.get_symbol (p. 59)

Reimplemented from **Unit** (p. 320).

4.8.2.6 def get_parent (*self*)

Returns the parent unit of the this unit.

Parameters:

self

Returns:

Parent unit.

4.8.2.7 def get_symbol (*self*)

Returns the symbol of this unit.

Parameters:

self

Returns:

Symbol of current unit.

4.8.2.8 def get_system_unit (*self*)

Returns the corresponding system unit. Since the parent unit is a system unit, this unit is supposed to be a system unit too. Therefore this function returns this instance.

Parameters:

self

Returns:

self

Reimplemented from **Unit** (p. 322).

4.8.2.9 def to_system_unit (*self*)

Get the operator to convert to the system unit. Since the parent unit is a system unit, this unit is supposed to be a system unit too. Therefore this function returns **operators.IDENTITY** (p. 8).

Parameters:

self

Returns:

operators.IDENTITY (p. 8)

See also:

operators.IDENTITY (p. 8)

Reimplemented from **Unit** (p. 324).

4.8.3 Member Data Documentation

4.8.3.1 __parentUnit__ = None `[static, private]`

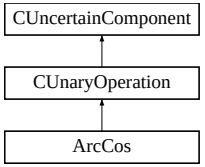
System unit that parents this unit.

4.8.3.2 __symbol__ = None `[static, private]`

Symbol for the alternate unit.

4.9 ArcCos Class Reference

Inheritance diagram for ArcCos::



4.9.1 Detailed Description

This class models the inverse cosine function.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.9.2 Member Function Documentation

4.9.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self
x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.9.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

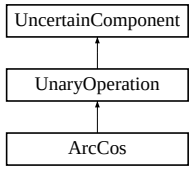
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.10 ArcCos Class Reference

Inheritance diagram for ArcCos::



4.10.1 Detailed Description

This class models the GUM-tree-nodes that take the Arcus Cosine of a silbling.

Public Member Functions

- def **__init__**
  Default constructor.
- def **arithmetic_check**
  Checks for undefined arguments.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = arccos(x)$ *then the resulting uncertainty is* $u(y) = -\frac{1}{\sqrt{1-x^2}}u(x)$.
- def **get_value**
  Returns the Arc Cosine of the silbling.

4.10.2 Member Function Documentation

4.10.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self

right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.10.2.2 def arithmetic_check (*self*)

Checks for undefined arguments.

Note:

The Arc Cosine is only defined within $[-1, 1]$.

Parameters:

self

Exceptions:

ArithmeticError If $x \notin [-1, 1]$.

Reimplemented from **UncertainComponent** (p. 302).

4.10.2.3 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.10.2.4 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = arccos(x)$ then the resulting uncertainty is $u(y) = -\frac{1}{\sqrt{1-x^2}}u(x)$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Exceptions:

ArithmeticError If $x^2 = 1$.

Reimplemented from **UncertainComponent** (p. 304).

4.10.2.5 def get_value (*self*)

Returns the Arc Cosine of the silbling.

Parameters:

self

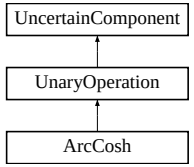
Returns:

A numeric value, representing the Arc Cosine of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.11 ArcCosh Class Reference

Inheritance diagram for ArcCosh::



4.11.1 Detailed Description

This class models the GUM-tree-nodes that take the inverse Hyperbolic Cosine.

Public Member Functions

- def **__init__**
  Default constructor.
- def **arithmetic_check**
  Checks for undefined arguments.
- def **equal_debug**
  A method that is only used for serialization checking.

- def **get_uncertainty**

  Returns the uncertainty of this node. Let the node represent the operation $y = arccosh(x)$ *then the resulting uncertainty is* $u(y) = \frac{1}{\sqrt{x-1}\sqrt{x+1}}u(x)$.

- def **get_value**

  Returns the Arc Cosine of the silbling.

4.11.2 Member Function Documentation

4.11.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self
right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.11.2.2 def arithmetic_check (*self*)

Checks for undefined arguments.

Note:

The inverse Hyperbolic Cosine is only defined within $(1, \infty]$.

Parameters:

self

Exceptions:

ArithmeticError If $x \notin (1, \infty]$.

Reimplemented from **UncertainComponent** (p. 302).

4.11.2.3 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self
other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.11.2.4 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = arccosh(x)$ then the resulting uncertainty is $u(y) = \frac{1}{\sqrt{x-1}\sqrt{x+1}}u(x)$.

Parameters:

self
component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.11.2.5 def get_value (*self*)

Returns the Arc Cosine of the silbling.

Parameters:

self

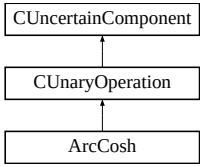
Returns:

A numeric value, representing the Arc Cosine of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.12 ArcCosh Class Reference

Inheritance diagram for ArcCosh::



4.12.1 Detailed Description

This class models the inverse hyperbolic cosine function.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.12.2 Member Function Documentation

4.12.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self
x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.12.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

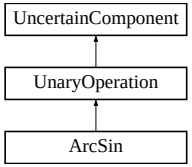
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.13 ArcSin Class Reference

Inheritance diagram for ArcSin::



4.13.1 Detailed Description

This class models the GUM-tree-nodes that take the Arc Sine of a silbling.

Public Member Functions

- def **__init__**
  Default constructor.
- def **arithmetic_check**
  Checks for undefined arguments.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = arcsin(x)$ *then the resulting uncertainty is* $u(y) = \frac{1}{\sqrt{1-x^2}}u(x)$.
- def **get_value**
  Returns the Arc Sine of the silbling.

4.13.2 Member Function Documentation

4.13.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self
right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.13.2.2 def arithmetic_check (*self*)

Checks for undefined arguments.

Note:

The Arc Sine is only defined within $[-1, 1]$.

Parameters:

self

Exceptions:

ArithmeticError If $x \notin [-1, 1]$.

Reimplemented from **UncertainComponent** (p. 302).

4.13.2.3 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.13.2.4 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = arcsin(x)$ then the resulting uncertainty is $u(y) = \frac{1}{\sqrt{1-x^2}} u(x)$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Exceptions:

ArithmeticError If $x^2 = 1$.

Reimplemented from **UncertainComponent** (p. 304).

4.13.2.5 def get_value (*self*)

Returns the Arc Sine of the silbling.

Parameters:

self

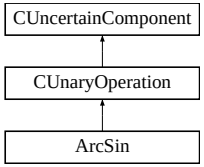
Returns:

A numeric value, representing the Arc Sine of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.14 ArcSin Class Reference

Inheritance diagram for ArcSin::



4.14.1 Detailed Description

This class models the inverse sine function.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.14.2 Member Function Documentation

4.14.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.14.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

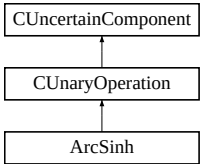
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.15 ArcSinh Class Reference

Inheritance diagram for ArcSinh::



4.15.1 Detailed Description

This class models the inverse hyperbolic sine function.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.15.2 Member Function Documentation

4.15.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.15.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

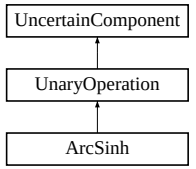
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.16 ArcSinh Class Reference

Inheritance diagram for ArcSinh::



4.16.1 Detailed Description

This class models the GUM-tree-nodes that take the inverse Hyperbolic Sine of a sibling.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**

Returns the uncertainty of this node. Let the node represent the operation $y = arcsinh(x)$ then the resulting uncertainty is $u(y) = \frac{1}{\sqrt{1+x^2}}u(x)$.

- def **get_value**

  Returns the inverse Hyperbolic Sine of a silbling.

4.16.2 Member Function Documentation

4.16.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self

right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.16.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.16.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = arcsinh(x)$ then the resulting uncertainty is $u(y) = \frac{1}{\sqrt{1+x^2}}u(x)$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.16.2.4 def get_value (*self*)

Returns the inverse Hyperbolic Sine of a silbling.

Parameters:

self

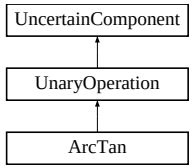
Returns:

A numeric value, representing the inverse Hyperbolic Sine of a silbling.

Reimplemented from **UncertainComponent** (p. 305).

4.17 ArcTan Class Reference

Inheritance diagram for ArcTan::



4.17.1 Detailed Description

This class models the GUM-tree-nodes that take the Arcus Tangent of a silbling.

Public Member Functions

- def **__init__**

  Default constructor.

- def **equal_debug**

  A method that is only used for serialization checking.

- def **get_uncertainty**

  Returns the uncertainty of this node. Let the node represent the operation $y = arcsin(x)$ then the resulting uncertainty is $u(y) = -\frac{1}{1+x^2}u(x)$.

- def **get_value**

  Returns the Arc Tangent of the silbling.

4.17.2 Member Function Documentation

4.17.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self

right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.17.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.17.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = arcsin(x)$ then the resulting uncertainty is $u(y) = -\frac{1}{1+x^2}u(x)$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.17.2.4 def get_value (*self*)

Returns the Arc Tangent of the silbling.

Parameters:

self

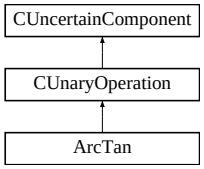
Returns:

A numeric value, representing the Arc Tangent of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.18 ArcTan Class Reference

Inheritance diagram for ArcTan::



4.18.1 Detailed Description

This class models the inverse tangent function.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.18.2 Member Function Documentation

4.18.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.18.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

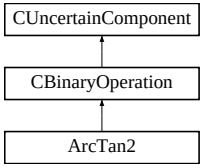
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.19 ArcTan2 Class Reference

Inheritance diagram for ArcTan2::



4.19.1 Detailed Description

This class models two-argument inverse tangent.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.19.2 Member Function Documentation

4.19.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.19.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

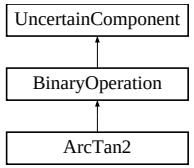
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.20 ArcTan2 Class Reference

Inheritance diagram for ArcTan2::



4.20.1 Detailed Description

This class models the inverse two-argument tangent.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.

- def **get_uncertainty**

  Returns the uncertainty of this node. Let the node represent the operation $y = x_1 + x_2$ then the resulting uncertainty is $u(y) = u(x_1) + u(x_2)$.

- def **get_value**

  Returns the sum of the silblings assigned.

4.20.2 Member Function Documentation

4.20.2.1 def __init__ (*self*, *left*, *right*)

Default constructor.

Parameters:

self

left Left silbling of this instance.

right Right silbling of this instance.

Reimplemented from **BinaryOperation** (p. 89).

4.20.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **BinaryOperation** (p. 89).

4.20.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = x_1 + x_2$ then the resulting uncertainty is $u(y) = u(x_1) + u(x_2)$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.20.2.4 def get_value (*self*)

Returns the sum of the silblings assigned.

Parameters:

self

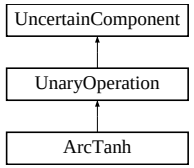
Returns:

A numeric value, representing the inverse two-argument tangent of the inputs.

Reimplemented from **UncertainComponent** (p. 305).

4.21 ArcTanh Class Reference

Inheritance diagram for ArcTanh::



4.21.1 Detailed Description

This class models the GUM-tree-nodes that take the inverse Hyperbolic Tangent of a silbling.

Public Member Functions

- def **__init__**

  Default constructor.

- def **arithmetic_check**

  Checks for undefined arguments.

- def **equal_debug**

  A method that is only used for serialization checking.

- def **get_uncertainty**

  Returns the uncertainty of this node. Let the node represent the operation $y = arctanh(x)$ then the resulting uncertainty is $u(y) = \frac{1}{1-x^2}u(x)$.

- def **get_value**

Returns the Arc Tangent of the silbling.

4.21.2 Member Function Documentation

4.21.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self

right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.21.2.2 def arithmetic_check (*self*)

Checks for undefined arguments.

Note:

The inverse Hyperbolic Tangent is only defined within $(-1, 1)$.

Parameters:

self

Exceptions:

ArithmeticError If $x \notin (-1, 1)$.

Reimplemented from **UncertainComponent** (p. 302).

4.21.2.3 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.21.2.4 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = arctanh(x)$ then the resulting uncertainty is $u(y) = \frac{1}{1-x^2}u(x)$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.21.2.5 def get_value (*self*)

Returns the Arc Tangent of the silbling.

Parameters:

self

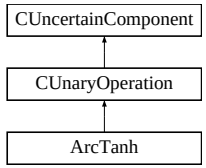
Returns:

A numeric value, representing the inverse hyperbolic Tangent of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.22 ArcTanh Class Reference

Inheritance diagram for ArcTanh::



4.22.1 Detailed Description

This class models the inverse hyperbolic tangent function.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.22.2 Member Function Documentation

4.22.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.22.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

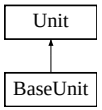
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.23 BaseUnit Class Reference

Inheritance diagram for BaseUnit::



4.23.1 Detailed Description

This class provides the interface to define and use base units.

A base unit is a unit that describes a single physical dimension. It can not be formed from other base units from other physical dimensions. Therefore, a base unit has to be unique. In order to ensure this, we do assign a unique symbol to each base unit.

Note:

Instances of this class can be serialized using pickle.

Public Member Functions

- def **__eq__**
  Check two if two base units are equal.
- def **__getstate__**
  Serialization using pickle.
- def **__init__**
  Default constructor.
- def **__setstate__**
  Deserialization using pickle.
- def **__str__**
  Return a string describing this unit.
- def **get_symbol**
  Return the symbol of this unit.
- def **get_system_unit**
  Get the corresponding system unit. Since it is a base unit, it returns itself.
- def **to_system_unit**
  *Get the operator to the system unit. Since it is a system unit, it returns **operators.IDENTITY** (p. 8).*

Static Private Attributes

- **__symbol__** = None
  *Unique symbol describing the **BaseUnit** (p. 83).*

4.23.2 Member Function Documentation

4.23.2.1 def __eq__ (*self*, *other*)

Check two if two base units are equal.

Two base units are equal if they have the same unit symbol.

Parameters:

self

other Another unit.

Reimplemented from **Unit** (p. 317).

4.23.2.2 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **Unit** (p. 317).

4.23.2.3 def __init__ (*self*, *symbol*)

Default constructor.

Assigns the desired symbol to the respective **BaseUnit** (p. 83) and checks if an other instance of a unit already exists that has the same symbol.

Parameters:

self

symbol A symbol identifying the **BaseUnit** (p. 83).

Exceptions:

UnitExistsException If a unit having the same symbol already exists.

See also:

AlternateUnit (p. 56)

4.23.2.4 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

Exceptions:

UnknownUnitException If the unit to be unpickled is not contained in the global repository __UNITS_MANAGER__.

See also:

UnitsManager (p. 329)
__UNITS_MANAGER__ (p. 23)

Reimplemented from **Unit** (p. 320).

4.23.2.5 def __str__ (*self*)

Return a string describing this unit.

Parameters:

self

Returns:

The symbol of this unit.

See also:

BaseUnit.get_symbol (p. 86)

Reimplemented from **Unit** (p. 320).

4.23.2.6 def get_symbol (*self*)

Return the symbol of this unit.

Parameters:

self

Returns:

The symbol of the **BaseUnit** (p. 83).

4.23.2.7 def get_system_unit (*self*)

Get the corresponding system unit. Since it is a base unit, it returns itself.

Parameters:

self

Returns:

The corresponding system unit.

See also:

Unit.get_system_unit (p. 322)

Reimplemented from **Unit** (p. 322).

4.23.2.8 def to_system_unit (*self*)

Get the operator to the system unit. Since it is a system unit, it returns **operators.IDENTITY** (p. 8).

Returns:

operators.IDENTITY (p. 8)

See also:

Unit.to_system_unit (p. 324)

Reimplemented from **Unit** (p. 324).

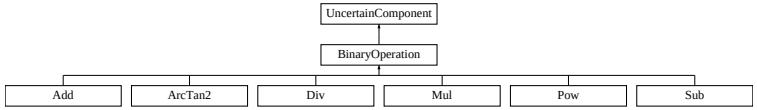
4.23.3 Member Data Documentation

4.23.3.1 __symbol__ = None `[static, private]`

Unique symbol describing the **BaseUnit** (p. 83).

4.24 BinaryOperation Class Reference

Inheritance diagram for BinaryOperation::



4.24.1 Detailed Description

The abstract base class for modelling binary operations. This class provides the abstract interface for GUM-tree-nodes that have two silblings.

Public Member Functions

- def **__getstate__**
  Serialization using pickle.
- def **__init__**
  Default constructor.
- def **__setstate__**
  Deserialization using pickle.
- def **depends_on**
  Get the components of uncertainty, that this class depends on.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_left**
  Return the left silbling.
- def **get_right**
  Return the right silbling.

Static Private Attributes

- **__left** = None
  The left silbling of the operation.
- **__right** = None
  The right silbling of the operation.

4.24.2 Member Function Documentation

4.24.2.1 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **UncertainComponent** (p. 295).

4.24.2.2 def __init__ (*self*, *left*, *right*)

Default constructor.

Attention:

If you extend this class call this constructor explicitly in order to initialize the silblings!

Parameters:

self

left Left silbling of this instance.

right Right silbling of this instance.

Reimplemented in **Add** (p. 50), **ArcTan2** (p. 79), **Mul** (p. 162), **Div** (p. 144), **Sub** (p. 242), and **Pow** (p. 174).

4.24.2.3 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

Reimplemented from **UncertainComponent** (p. 299).

4.24.2.4 def depends_on (*self*)

Get the components of uncertainty, that this class depends on.

Returns:

A list of the components of uncertainty.

Reimplemented from **UncertainComponent** (p. 303).

4.24.2.5 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UncertainComponent** (p. 303).

Reimplemented in **Add** (p. 51), **ArcTan2** (p. 79), **Mul** (p. 162), **Div** (p. 144), **Sub** (p. 243), and **Pow** (p. 174).

4.24.2.6 def get_left (*self*)

Return the left silbling.

Returns:

The left silbling.

4.24.2.7 def get_right (*self*)

Return the right silbling.

Returns:

The right silbling.

4.24.3 Member Data Documentation

4.24.3.1 __left = None [static, private]

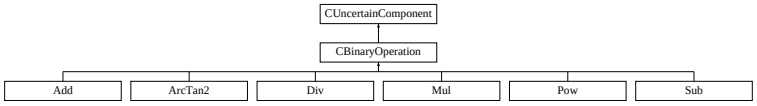
The left silbling of the operation.

4.24.3.2 __right = None [static, private]

The right silbling of the operation.

4.25 CBinaryOperation Class Reference

Inheritance diagram for CBinaryOperation::



4.25.1 Detailed Description

This abstract class models a binary operation.

Public Member Functions

- def **__init__**
  The default constructor.
- def **depends_on**
  *Get the instances of **CUncertainInput** (p. 133) that this instance depends on.*
- def **get_left**
  Get the left sibling of this operation.
- def **get_right**
  Get the right sibling of this operation.

Private Attributes

- **__left**
- **__right**

4.25.2 Member Function Documentation

4.25.2.1 def __init__ (*self*, *left*, *right*)

The default constructor.

Parameters:

self
left The left sibling sibling of this operation.
right The right sibling sibling of this operation.

4.25.2.2 def depends_on (*self*)

Get the instances of **CUncertainInput** (p. 133) that this instance depends on.

Parameters:

self

Returns:

A list containing the instances of **CUncertainInput** (p. 133) that this instance depends on.

Reimplemented from **CUncertainComponent** (p. 128).

4.25.2.3 def get_left (*self*)

Get the left sibling of this operation.

Parameters:

self

Returns:

The sibling

4.25.2.4 def get_right (*self*)

Get the right sibling of this operation.

Parameters:

self

Returns:

The sibling

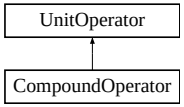
4.25.3 Member Data Documentation

4.25.3.1 __left [private]

4.25.3.2 __right [private]

4.26 CompoundOperator Class Reference

Inheritance diagram for CompoundOperator::



4.26.1 Detailed Description

Compound Operator.

This class is used to generate compound operators (i.e. by calling the concat method of the **UnitOperator** (p. 326)).

Note:

Instances of this class can be serialized using pickle.

Public Member Functions

- def **__eq__**
  Test for equality.
- def **__getstate__**
  Serialization using pickle.
- def **__init__**
  Default Constructor.
- def **__invert__**
  Invert the current operation.
- def **__setstate__**
  Deserialization using pickle.
- def **__str__**
  Represent this operation by a string.
- def **convert**
  Convert a value.
- def **is_linear**
  Check if the Operator is linear.

Private Attributes

- **__firstOperator__**
- **__secondOperator__**

4.26.2 Member Function Documentation

4.26.2.1 def __eq__ (*self*, *other*)

Test for equality.

Parameters:

self

other Another **UnitOperator** (p. 326).

Reimplemented from **UnitOperator** (p. 327).

4.26.2.2 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **UnitOperator** (p. 327).

4.26.2.3 def __init__ (*self*, *firstOp*, *secondOp*)

Default Constructor.

For example let the secondOp be $g(x)$ and the firstOp be $f(x)$ then the compound Operator models $f(g(x))$.

Parameters:

self

firstOp The operator that is performed at first.

secondOp The operator that is performed at last.

4.26.2.4 def __invert__ (*self*)

Invert the current operation.

This method returns the inverse Operation of the current operation. Since this Operation is based on two Operations the operations are inverted in the reverse order. For example let this Operator model $y = f(g(x))$ the inverse Operator models $x = g^{-1}(f^{-1}(y))$.

Parameters:

self

Returns:

The inverse operation of the current operation.

Reimplemented from **UnitOperator** (p. 327).

4.26.2.5 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

Reimplemented from **UnitOperator** (p. 328).

4.26.2.6 def __str__ (*self*)

Represent this operation by a string.

Parameters:

self

Returns:

A string describing this operation.

Reimplemented from **UnitOperator** (p. 328).

4.26.2.7 def convert (*self*, *value*)

Convert a value.

This method performs the desired operation on an absolute value.

Parameters:

self The current instance of this class.

value The value to convert.

Returns:

the converted value

Reimplemented from **UnitOperator** (p. 329).

4.26.2.8 def is_linear (*self*)

Check if the Operator is linear.

This operator is linear if the underlying operators are linear.

Parameters:

self The current instance of this class.

Returns:

`True` if both underlying operators are linear.

Reimplemented from **UnitOperator** (p. 329).

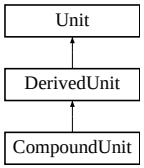
4.26.3 Member Data Documentation

4.26.3.1 __firstOperator__ `[private]`

4.26.3.2 __secondOperator__ `[private]`

4.27 CompoundUnit Class Reference

Inheritance diagram for CompoundUnit::



4.27.1 Detailed Description

This class provides an interface for describing compound units. The units forming a compound unit have to describe the same physical dimension. For example time $[hour : min : second]$.

Note:

Instances of this class can be serialized using pickle.

Public Member Functions

- def **__eq__**

  This function checks if two compound units are equal. Two compound units are equal, if they have equal first and next units.

- def **__getstate__**

  Serialization using pickle.

- def **__init__**

  Default constructor. Both arguments have to describe the same physical dimension and they have to have the same system unit.

- def **__setstate__**

  Deserialization using pickle.

- def **__str__**

Print the current unit. This function returns a string of the form `first:next`.

- def **get_first**

  Get the first unit.

- def **get_next**

  Get the next unit.

- def **get_system_unit**

  Returns the corresponding system unit.

- def **to_system_unit**

  Get the operator to convert to the system unit. We assume that the operator of the first element of this compound unit performs the conversion correctly.

Static Private Attributes

- **__first__** = None

  The first unit.

- **__next__** = None

  The next unit.

4.27.2 Member Function Documentation

4.27.2.1 def __eq__ (*self*, *other*)

This function checks if two compound units are equal. Two compound units are equal, if they have equal first and next units.

Attention:

The order of the first and next units matters (i.e. $hh : mm \neq mm : hh$)!

Parameters:

self

other Another compound unit to compare to.

Returns:

True, if the units are equal.

Reimplemented from **Unit** (p. 317).

4.27.2.2 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **Unit** (p. 317).

4.27.2.3 def __init__ (*self*, *firstUnit*, *nextUnit*)

Default constructor. Both arguments have to describe the same physical dimension and they have to have the same system unit.

Parameters:

self

firstUnit The first unit.

nextUnit The unit to attach to the first unit.

Exceptions:

TypeError If the units describe different dimensions.

4.27.2.4 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

Reimplemented from **Unit** (p. 320).

4.27.2.5 def __str__ (*self*)

Print the current unit. This function returns a string of the form `first:next`.

Parameters:

self

Returns:

A string describing this unit.

Reimplemented from **Unit** (p. 320).

4.27.2.6 def get_first (*self*)

Get the first unit.

Parameters:

self

Returns:

The first unit of this compound unit.

4.27.2.7 def get_next (*self*)

Get the next unit.

Parameters:

self

Returns:

The first unit of this compound unit.

4.27.2.8 def get_system_unit (*self*)

Returns the corresponding system unit.

Note:

All units forming this unit have the same system unit.

Returns:

The corresponding system unit.

Reimplemented from **Unit** (p. 322).

4.27.2.9 def to_system_unit (*self*)

Get the operator to convert to the system unit. We assume that the operator of the first element of this compound unit performs the conversion correctly.

Parameters:

self

Returns:

The operator to the system unit of the first element

See also:

CompoundUnit.__first__ (p. 100)

Reimplemented from **Unit** (p. 324).

4.27.3 Member Data Documentation

4.27.3.1 __first__ = None [static, private]

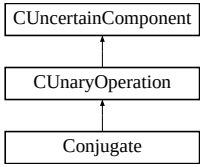
The first unit.

4.27.3.2 __next__ = None [static, private]

The next unit.

4.28 Conjugate Class Reference

Inheritance diagram for Conjugate::



4.28.1 Detailed Description

This class models taking the negative of a complex value.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.28.2 Member Function Documentation

4.28.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.28.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.29 Context Class Reference

4.29.1 Detailed Description

This class provides the context for an uncertainty evaluation. It maintains the correlation between the inputs and can be used to evaluate the combined standard uncertainty, as shown below. Let your model be $y = f(x_1, x_2, \ldots, x_N)$, then $u_c^2(y) = \sum_{i=1}^{N} \left(\frac{\delta f}{\delta x_i}\right)^2 u^2(x_i) + 2\sum_{i=1}^{N}\sum_{j=i+1}^{N} \frac{\delta f}{\delta x_i}\frac{\delta f}{\delta x_j} u(x_i, x_j)$.

Public Member Functions

- def **__init__**

  This method initializes the correlation matrix of this context.

- def **dof**

  This method calculates the effective degrees of freedom using the Welch-Satterthwaite formulae: $\nu_{eff} = \frac{u_c^4(y)}{\sum_{i=1}^{N} \frac{\left(\frac{\delta f}{\delta x_i}\right)^4 u^4(x_i)}{\nu_i}}$ *Where* $u_c(y)$ *is the combined standard uncertainty,* ν_i *is the degrees of freedom of the input* x_i.

- def **get_correlation**

  This method returns the correlation coefficient $r(x_1, x_2)$ *of two inputs. Where* $r(x_1, x_2) = \frac{u(x_1, x_2)}{u(x_1)u(x_2)}$. *If no correlation has been defined before, this method returns* `0.0`, *except for* $x_1 = x_2$. *In the last case this method returns* `1.0`.

- def **set_correlation**

  This method sets the correlation coefficient $r(x_1, x_2)$ *of two inputs. Where* $r(x_1, x_2) = \frac{u(x_1, x_2)}{u(x_1)u(x_2)}$.

- def **uncertainty**

  This method returns the combined standard uncertainty of an uncertain value.

- def **value_of**

Private Attributes

- **__correlationMatrix**

4.29.2 Member Function Documentation

4.29.2.1 def __init__ (*self*)

This method initializes the correlation matrix of this context.

Note:

You may use the same uncertain components in different contexts. Thus, you could maintain various correlation models.

Parameters:

self

4.29.2.2 def dof (*self*, *component*)

This method calculates the effective degrees of freedom using the Welch-Satterthwaite formulae: $\nu_{eff} = \frac{u_c^4(y)}{\sum_{i=1}^{N} \frac{\left(\frac{\delta f}{\delta x_i}\right)^4 u^4(x_i)}{\nu_i}}$ Where $u_c(y)$ is the combined standard uncertainty, ν_i is the degrees of freedom of the input x_i.

Note:

The result of this method may be infinite. Since there is no standard procedure in python to declare infinity, we use our own constant for it.

See also:

arithmetic.INFINITY (p. 4) Our infinity constant.

Parameters:

self

component The component of uncertainty.

Returns:

The effective degrees of freedom ν_{eff}.

4.29.2.3 def get_correlation (*self*, *firstItem*, *secondItem*)

This method returns the correlation coefficient $r(x_1, x_2)$ of two inputs. Where $r(x_1, x_2) = \frac{u(x_1,x_2)}{u(x_1)u(x_2)}$. If no correlation has been defined before, this method returns `0.0`, except for $x_1 = x_2$. In the last case this method returns `1.0`.

Note:

This libary assumes symmetry of correlation (i.e. $r(x_1, x_2) = r(x_2, x_1)$).

Parameters:

self

firstItem Is x_1 as denoted above.

secondItem Is x_2 as denoted above.

4.29.2.4 def set_correlation (*self*, *firstItem*, *secondItem*, *corr*)

This method sets the correlation coefficient $r(x_1, x_2)$ of two inputs. Where $r(x_1, x_2) = \frac{u(x_1,x_2)}{u(x_1)u(x_2)}$.

Note:

This libary assumes symmetry of correlation (i.e. $r(x_1, x_2) = r(x_2, x_1)$).

Attention:

If the arguments are identical, this method has no effect.

Parameters:

self

firstItem Is x_1 as denoted above.

secondItem Is x_2 as denoted above.

corr The correlation as described by $r(x_1, x_2)$.

4.29.2.5 def uncertainty (*self*, *component*)

This method returns the combined standard uncertainty of an uncertain value.

Parameters:

self

component The component of uncertainty to evaluate.

Returns:

The standard uncertainty.

4.29.2.6 def value_of (*self*, *component*)

Assign the current context to the given component.

Attention:

This method is only useful in combination with **UncertainComponent.__str__** (p. 299). The context assigned is not passed to operations performed on `component`.

See also:

UncertainComponent.__str__ (p. 299)

Parameters:

self

component The component to which the context should be attached.

Returns:

component having the context assigned.

4.29.3 Member Data Documentation

4.29.3.1 __correlationMatrix [private]

4.30 Context Class Reference

4.30.1 Detailed Description

This class provides a context for complex-valued uncertainty evaluations. It manages the correlation coefficients and is able to evaluate the effective degrees of freedom.

Public Member Functions

- def **__init__**
  The default constructor. It initializes the dictionary of correlation matrices.
- def **constant**
  This is a factory method for generating constans for uncertainty evaluations.
- def **dof**
  Calculate the effective degrees of freedom of the argument.
- def **gaussian**
  This is a factory method for generating uncertain inputs that have a Gaussian distribution (i.e. bivariate Normal Distribution).

- def **get_correlation**

  Get the correlation of two input arguments.

- def **set_correlation**

  This method sets the correlation coefficients of two input arguments.

- def **uncertainty**

  Get the combined standard uncertainty of a complex-valued component of uncertainty.

Private Member Functions

- def **__check_cmatrix**

  Helper function to verify matrices of corellation coefficients.

Private Attributes

- **__correlation**

Static Private Attributes

- tuple **__check_cmatrix** = staticmethod(**__check_cmatrix**)

4.30.2 Member Function Documentation

4.30.2.1 def __check_cmatrix (*matrix*) [private]

Helper function to verify matrices of corellation coefficients.

Parameters:

matrix The matrix to check.

Exceptions:

TypeError If the argument is invalid

4.30.2.2 def __init__ (*self*)

The default constructor. It initializes the dictionary of correlation matrices.

Parameters:

self

4.30.2.3 def constant (*self*, *val*)

This is a factory method for generating constans for uncertainty evaluations.

4.30.2.4 def dof (*self*, *c*)

Calculate the effective degrees of freedom of the argument.

Parameters:

self

c The component of uncertainty.

Returns:

The number of effective degress of freedom.

Attention:

The result may me infinite if any of the inputs has an infinite DOF. In this case this method returns **`arithmetic.INFINITY`** (p. 4).

See also:

arithmetic.INFINITY (p. 4)

4.30.2.5 def gaussian (*self*, *val*, *u_r*, *u_i*, *dof* = arithmetic.INFINITY, *matrix* = `numpy.matrix([[1, 0]])`

This is a factory method for generating uncertain inputs that have a Gaussian distribution (i.e. bivariate Normal Distribution).

Parameters:

self

val The complex value of the input.

u_r The uncertainty of the real part.

u_i The uncertainty of the imaginary part.

dof The degrees of freedom of the variable.

matrix The matrix of correlation coefficients.

4.30.2.6 def get_correlation (*self*, *c1*, *c2*)

Get the correlation of two input arguments.

Parameters:

self

c1 The first **CUncertainInput** (p. 133)

c2 The second **CUncertainInput** (p. 133)

Returns:

The matrix of correlation coefficients.

4.30.2.7 def set_correlation (*self*, *c1*, *c2*, *matrix*)

This method sets the correlation coefficients of two input arguments.

Parameters:

self

c1 The first **CUncertainInput** (p. 133)

c2 The second **CUncertainInput** (p. 133)

matrix The matrix of correlation coefficients

4.30.2.8 def uncertainty (*self*, *c*)

Get the combined standard uncertainty of a complex-valued component of uncertainty.

Parameters:

self

c The component of uncertainty.

Returns:

The matrix expressing the combined standard uncertainty.

Attention:

If the argument is an instance of Quantitiy having the unit [u] then the uncertainty, expressed as covariance matrix has [u^2].

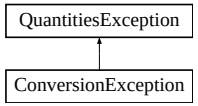
4.30.3 Member Data Documentation

4.30.3.1 tuple __check_cmatrix = staticmethod(__check_cmatrix) [static, private]

4.30.3.2 __correlation [private]

4.31 ConversionException Class Reference

Inheritance diagram for ConversionException::



4.31.1 Detailed Description

General exception that is raised whenever a unit conversion fails.

See also:

units.Unit.to_system_unit (p. 324)
units.Unit.get_operator_to (p. 322)
operators.UnitOperator (p. 326)

Public Member Functions

- def **__init__**
  Default constructor.
- def **__str__**
  Returns a string describing this exception.

Private Attributes

- **__unit__**

4.31.2 Member Function Documentation

4.31.2.1 def __init__ (*self*, *unit*, *args*)

Default constructor.

Parameters:

self

unit Instance of a unit that raised the exception.

args Additional arguments of this exception.

4.31.2.2 def __str__ (*self*)

Returns a string describing this exception.

Parameters:

self

Returns:

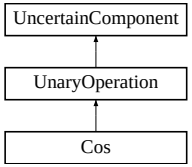
A string that describes the exception.

4.31.3 Member Data Documentation

4.31.3.1 __unit__ [private]

4.32 Cos Class Reference

Inheritance diagram for Cos::



4.32.1 Detailed Description

This class models the GUM-tree-nodes that take the Cosine of a silbling.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = cos(x)$ then the resulting uncertainty is $u(y) = -sin(x) \times u(x)$.
- def **get_value**
  Returns the Cosine of the silbling.

4.32.2 Member Function Documentation

4.32.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self
right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.32.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self
other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.32.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = cos(x)$ then the resulting uncertainty is $u(y) = -sin(x) \times u(x)$.

Parameters:

self
component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.32.2.4 def get_value (*self*)

Returns the Cosine of the silbling.

Parameters:

self

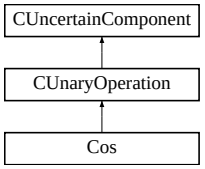
Returns:

A numeric value, representing the Cosine of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.33 Cos Class Reference

Inheritance diagram for Cos::



4.33.1 Detailed Description

This class models the cosine function.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.33.2 Member Function Documentation

4.33.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.33.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

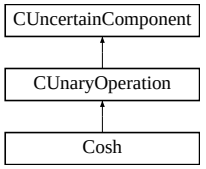
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.34 Cosh Class Reference

Inheritance diagram for Cosh::



4.34.1 Detailed Description

This class models the hyperbolic cosine function.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.34.2 Member Function Documentation

4.34.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.34.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

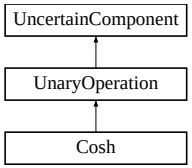
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.35 Cosh Class Reference

Inheritance diagram for Cosh::



4.35.1 Detailed Description

This class models the GUM-tree-nodes that take the Hyperbolic Cosine of a silbling.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.

- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = cosh(x)$ then the resulting uncertainty is $u(y) = sinh(x)u(x)$.
- def **get_value**
  Returns the Hyperbolic Cosine of the silbling.

4.35.2 Member Function Documentation

4.35.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self
right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.35.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self
other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.35.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = cosh(x)$ then the resulting uncertainty is $u(y) = sinh(x)u(x)$.

Parameters:

self
component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.35.2.4 def get_value (*self*)

Returns the Hyperbolic Cosine of the silbling.

Parameters:

self

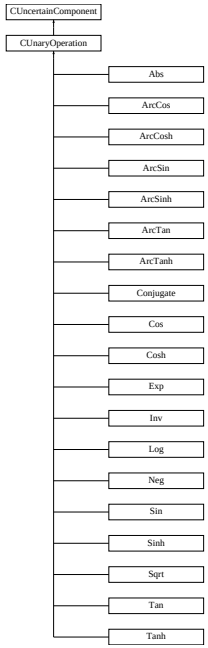
Returns:

A numeric value, representing the Hyperbolic Cosine of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.36 CUnaryOperation Class Reference

Inheritance diagram for CUnaryOperation::



4.36.1 Detailed Description

This abstract class models an unary operation.

Public Member Functions

- def **__init__**
  The default constructor.
- def **depends_on**
  Get the instances of ***CUncertainInput*** *(p. 133) that this instance depends on.*
- def **get_sibling**
  Get the sibling of this operation.

Private Attributes

- **__sibling**

4.36.2 Member Function Documentation

4.36.2.1 def __init__ (*self*, *sibling*)

The default constructor.

Parameters:

self

sibling The sibling of this operation.

4.36.2.2 def depends_on (*self*)

Get the instances of **CUncertainInput** (p. 133) that this instance depends on.

Parameters:

self

Returns:

A list containing the instances of **CUncertainInput** (p. 133) that this instance depends on.

Reimplemented from **CUncertainComponent** (p. 128).

4.36.2.3 def get_sibling (*self*)

Get the sibling of this operation.

Parameters:

self

Returns:

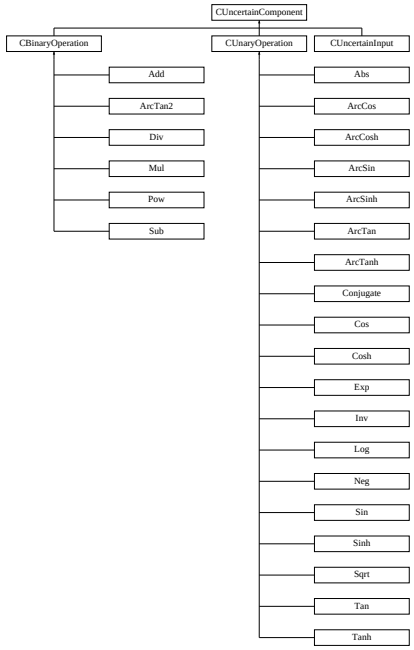
The sibling

4.36.3 Member Data Documentation

4.36.3.1 __sibling [private]

4.37 CUncertainComponent Class Reference

Inheritance diagram for CUncertainComponent::



4.37.1 Detailed Description

This is the abstract super class of all complex valued uncertain components. Despite defining the interface for complex valued uncertain components, it also provides a set of factory methods that act as an interface for `numpy`.

Public Member Functions

- def **__abs__**

  Return the absolute value of this instance. Let this instance be $\mathbf{z} = x + jy$ *then this method returns* $\sqrt{x^2 + y^2}$.

- def **__add__**

  Add *(*p. 48*) another instance to this instance.*

- def **__coerce__**

  Implementation of coercion rules.

- def **__div__**

  Divide this instance by another instance.

- def **__invert__**

  Get the inverse of this instance. Let this instance be x *then this method returns* x^{-1}.

- def **__mul__**

  Multiply another instance with this instance.

- def **__neg__**

  Negate this instance.

- def **__pow__**

  Raise this instance to the power of the argument.

- def **__radd__**

  Add *(*p. 48*) another instance to this instance.*

- def **__rdiv__**

  Divide this instance by another instance.

- def **__rmul__**

  Multiply another instance with this instance.

- def **__rpow__**

  Raise this instance to the power of the argument.

- def **__rsub__**

Subtract another instance from this instance.

- def **__str__**

  This method prints the component of uncertainty.

- def **__sub__**

  Subtract another instance from this instance.

- def **arccos**

  Get the inverse cosine of this instance. Let this instance be x then this method returns $\cos^{-1}(x)$.

- def **arccosh**

  Get the inverse hyperbolic cosine of this instance. Let this instance be x then this method returns $\cosh^{-1}(x)$.

- def **arcsin**

  Get the inverse sine of this instance. Let this instance be x then this method returns $\sin^{-1}(x)$.

- def **arcsinh**

  Get the inverse hyperbolic sine of this instance. Let this instance be x then this method returns $\sinh^{-1}(x)$.

- def **arctan**

  Get the inverse tangent of this instance. Let this instance be x then this method returns $\tan^{-1}(x)$.

- def **arctan2**

  Get the two-argument inverse tangent of this instance. Let this instance be x then this method returns $\tan^{-1}(x)$.

- def **arctanh**

  Get the inverse hyperbolic tangent of this instance. Let this instance be x then this method returns $\tanh^{-1}(x)$.

- def **conjugate**

  Get the conjuagte complex value of this instance.

- def **cos**

  Get the cosine of this instance. Let this instance be x then this method returns $\cos(x)$.

- def **cosh**

  Get the hyperbolic cosine of this instance. Let this instance be x then this method returns $\cosh(x)$.

- def **depends_on**

*This abstact method should return the set of **CUncertainInput** (*p. *133)* *instances, on which this instance depends on.*

- def **exp**

  Get the exponential of this instance. Let this instance be x then this method returns e^x.

- def **fabs**

  Return the absolute value of this instance. Let this instance be $\mathbf{z} = x + jy$ then this method returns $\sqrt{x^2 + y^2}$.

- def **get_a_value**

  This method returns the value of this component as a 2x2-matrix.

- def **get_context**

  This returns the assigned context of the component. This context is only needed for evaluating __str__.

- def **get_uncertainty**

  This abstact method should return the partial derivate of this component with respect to the input x.

- def **get_value**

  This abstract method should return the complex value of this component.

- def **hypot**

  Calculate the hypothenusis of this and another complex-valued argument.

- def **log**

  Get the natural logarithm of this instance. Let this instance be x then this method returns $\ln(x)$.

- def **log10**

  Get the decadic logarithm of this instance. Let this instance be x then this method returns $\log_{10}(x)$.

- def **log2**

  Get the binary logarithm of this instance. Let this instance be x then this method returns $\log_2(x)$.

- def **set_context**

  This assigns a context to the component. This context is only needed for evaluating __str__.

- def **sin**

  Get the sine of this instance. Let this instance be x then this method returns $\sin(x)$.

- def **sinh**

Get the hyperbolic sine of this instance. Let this instance be x then this method returns $\sinh(x)$.

- def **sqrt**

  Get the square-root of this instance. Let this instance be x then this method returns $\sqrt{x}$.

- def **square**

  Get the square of this instance. Let this instance be x then this method returns $x \cdot x$.

- def **tan**

  Get the tangent of this instance. Let this instance be x then this method returns $\tan(x)$.

- def **tanh**

  Get the hyperbolic tangent of this instance. Let this instance be x then this method returns $\tanh(x)$.

- def **value_of**

  This factory method converts the argument to a complex uncertain value.

Static Public Attributes

- tuple **value_of** = staticmethod(**value_of**)

Private Attributes

- **__context**

4.37.2 Member Function Documentation

4.37.2.1 def __abs__ (*self*)

Return the absolute value of this instance. Let this instance be $\mathbf{z} = x + jy$ then this method returns $\sqrt{x^2 + y^2}$.

Parameters:

self

Returns:

The absolute value of this instance.

4.37.2.2 def __add__ (*self*, *y*)

Add (p. 48) another instance to this instance.

Parameters:

self

y Another uncertain value.

Returns:

The sum of this instance and the other instance.

4.37.2.3 def __coerce__ (*self*, *other*)

Implementation of coercion rules.

See also:

Coercion - The page describing the coercion rules.

4.37.2.4 def __div__ (*self*, *y*)

Divide this instance by another instance.

Parameters:

self

y Another uncertain value.

Returns:

The result of the respective operation.

4.37.2.5 def __invert__ (*self*)

Get the inverse of this instance. Let this instance be x then this method returns x^{-1}.

Parameters:

self

Returns:

The inverse of this instance.

4.37.2.6 def __mul__ (*self*, *y*)

Multiply another instance with this instance.

Parameters:

self

y Another uncertain value.

Returns:

The product of this instance and the other instance.

4.37.2.7 def __neg__ (*self*)

Negate this instance.

Parameters:

self

Returns:

The negative of this instance.

4.37.2.8 def __pow__ (*self*, *y*)

Raise this instance to the power of the argument.

Parameters:

self

y Another uncertain value.

Returns:

The result of the respective operation.

4.37.2.9 def __radd__ (*self*, *y*)

Add (p. 48) another instance to this instance.

Parameters:

self

y Another uncertain value.

Returns:

The sum of this instance and the other instance.

4.37.2.10 def __rdiv__ (*self*, *y*)

Divide this instance by another instance.

Parameters:

self

y Another uncertain value.

Returns:

The result of the respective operation.

4.37.2.11 def __rmul__ (*self*, *y*)

Multiply another instance with this instance.

Parameters:

self

y Another uncertain value.

Returns:

The product of this instance and the other instance.

4.37.2.12 def __rpow__ (*self*, *y*)

Raise this instance to the power of the argument.

Parameters:

self

y Another uncertain value.

Returns:

The result of the respective operation.

4.37.2.13 def __rsub__ (*self*, *y*)

Subtract another instance from this instance.

Parameters:

self

y Another uncertain value.

Returns:

The difference of this instance and the other instance.

4.37.2.14 def __str__ (*self*)

This method prints the component of uncertainty.

Parameters:

self

Returns:

A string describing this component

4.37.2.15 def __sub__ (*self*, *y*)

Subtract another instance from this instance.

Parameters:

self

y Another uncertain value.

Returns:

The difference of this instance and the other instance.

4.37.2.16 def arccos (*self*)

Get the inverse cosine of this instance. Let this instance be x then this method returns $\cos^{-1}(x)$.

Parameters:

self

Returns:

The inverse cosine of this instance.

4.37.2.17 def arccosh (*self*)

Get the inverse hyperbolic cosine of this instance. Let this instance be x then this method returns $\cosh^{-1}(x)$.

Parameters:

self

Returns:

The inverse hyperbolic cosine of this instance.

4.37.2.18 def arcsin (*self*)

Get the inverse sine of this instance. Let this instance be x then this method returns $\sin^{-1}(x)$.

Parameters:

self

Returns:

The inverse sine of this instance.

4.37.2.19 def arcsinh (*self*)

Get the inverse hyperbolic sine of this instance. Let this instance be x then this method returns $\sinh^{-1}(x)$.

Parameters:

self

Returns:

The inverse hyperbolic sine of this instance.

4.37.2.20 def arctan (*self*)

Get the inverse tangent of this instance. Let this instance be x then this method returns $\tan^{-1}(x)$.

Parameters:

self

Returns:

The inverse tangent of this instance.

4.37.2.21 def arctan2 (*self*, *y*)

Get the two-argument inverse tangent of this instance. Let this instance be x then this method returns $\tan^{-1}(x)$.

Parameters:

self

y Another component of uncertainty.

Returns:

The two-argument inverse tangent of this instance.

4.37.2.22 def arctanh (*self*)

Get the inverse hyperbolic tangent of this instance. Let this instance be x then this method returns $\tanh^{-1}(x)$.

Parameters:

self

Returns:

The inverse hyperbolic tangent of this instance.

4.37.2.23 def conjugate (*self*)

Get the conjuagte complex value of this instance.

Parameters:

self

Returns:

the conjuagte complex value of this instance.

4.37.2.24 def cos (*self*)

Get the cosine of this instance. Let this instance be x then this method returns $\cos(x)$.

Parameters:

self

Returns:

The cosine of this instance.

4.37.2.25 def cosh (*self*)

Get the hyperbolic cosine of this instance. Let this instance be x then this method returns $\cosh(x)$.

Parameters:

self

Returns:

The hyperbolic cosine of this instance.

4.37.2.26 def depends_on (*self*)

This abstact method should return the set of **CUncertainInput** (p. 133) instances, on which this instance depends on.

Parameters:

self

Returns:

A list of CUncertainInputs this instance depends on.

Attention:

This method needs to be overriden to have an effect.

Reimplemented in **CUncertainInput** (p. 135), **CUnaryOperation** (p. 116), and **CBinaryOperation** (p. 91).

4.37.2.27 def exp (*self*)

Get the exponential of this instance. Let this instance be x then this method returns e^x.

Parameters:

self

Returns:

The exponential value of this instance.

4.37.2.28 def fabs (*self*)

Return the absolute value of this instance. Let this instance be $\mathbf{z} = x + jy$ then this method returns $\sqrt{x^2 + y^2}$.

Parameters:

self

Returns:

The absolute value of this instance.

4.37.2.29 def get_a_value (*self*)

This method returns the value of this component as a 2x2-matrix.

Parameters:

self

Returns:

The complex value of this component in matrix form.

Reimplemented in **CUncertainInput** (p. 135).

4.37.2.30 def get_context (*self*)

This returns the assigned context of the component. This context is only needed for evaluating __str__.

Returns:

c The **Context** (p. 104) of the component or `None`.

4.37.2.31 def get_uncertainty (*self*, *x*)

This abstact method should return the partial derivate of this component with respect to the input `x`.

Parameters:

self

x An uncertain input.

Returns:

The uncertainty of this component with respect to the input.

Reimplemented in **CUncertainInput** (p. 136), **Exp** (p. 147), **Log** (p. 156), **Sqrt** (p. 239), **Sin** (p. 235), **Cos** (p. 111), **Tan** (p. 247), **ArcSin** (p. 70), **ArcCos** (p. 61), **ArcTan** (p. 76), **Sinh** (p. 236), **Cosh** (p. 112), **Tanh** (p. 250), **ArcSinh** (p. 71), **ArcCosh** (p. 67), **ArcTanh** (p. 83), **Abs** (p. 46), **Conjugate** (p. 100), **Neg** (p. 170), **Inv** (p. 152), **Add** (p. 49), **Sub** (p. 244), **Mul** (p. 161), **Div** (p. 142), **Pow** (p. 176), and **ArcTan2** (p. 77).

4.37.2.32 def get_value (*self*)

This abstract method should return the complex value of this component.

Parameters:

self

Returns:

The value of this component.

Attention:

This method needs to be overriden to have an effect.

Reimplemented in **CUncertainInput** (p. 136), **Exp** (p. 148), **Log** (p. 156), **Sqrt** (p. 239), **Sin** (p. 235), **Cos** (p. 112), **Tan** (p. 247), **ArcSin** (p. 70), **ArcCos** (p. 61), **ArcTan** (p. 77), **Sinh** (p. 236), **Cosh** (p. 113), **Tanh** (p. 250), **ArcSinh** (p. 72), **ArcCosh** (p. 67), **ArcTanh** (p. 83), **Abs** (p. 46), **Conjugate** (p. 101), **Neg** (p. 171), **Inv** (p. 152), **Add** (p. 49), **Sub** (p. 244), **Mul** (p. 161), **Div** (p. 142), **Pow** (p. 176), and **ArcTan2** (p. 78).

4.37.2.33 def hypot (*self*, *y*)

Calculate the hypothenusis of this and another complex-valued argument.

Parameters:

self

y another component of uncertainty.

Returns:

$\sqrt{x^2 + y^2}$

4.37.2.34 def log (*self*)

Get the natural logarithm of this instance. Let this instance be x then this method returns $\ln(x)$.

Parameters:

self

Returns:

The natural logarithm of this instance.

4.37.2.35 def log10 (*self*)

Get the decadic logarithm of this instance. Let this instance be x then this method returns $\log_{10}(x)$.

Parameters:

self

Returns:

The decadic logarithm of this instance.

4.37.2.36 def log2 (*self*)

Get the binary logarithm of this instance. Let this instance be x then this method returns $\log_2(x)$.

Parameters:

self

Returns:

The binary logarithm of this instance.

4.37.2.37 def set_context (*self*, *c*)

This assigns a context to the component. This context is only needed for evaluating __str__.

Parameters:

self

c An instance of **Context** (p. 104)

4.37.2.38 def sin (*self*)

Get the sine of this instance. Let this instance be x then this method returns $\sin(x)$.

Parameters:

self

Returns:

The sine of this instance.

4.37.2.39 def sinh (*self*)

Get the hyperbolic sine of this instance. Let this instance be x then this method returns $\sinh(x)$.

Parameters:

self

Returns:

The hyperbolic sine of this instance.

4.37.2.40 def sqrt (*self*)

Get the square-root of this instance. Let this instance be x then this method returns $\sqrt{x}$.

Parameters:

self

Returns:

The square-root of this instance.

4.37.2.41 def square (*self*)

Get the square of this instance. Let this instance be x then this method returns $x \cdot x$.

Parameters:

self

Returns:

The square of this instance.

4.37.2.42 def tan (*self*)

Get the tangent of this instance. Let this instance be x then this method returns $\tan(x)$.

Parameters:

self

Returns:

The tangent of this instance.

4.37.2.43 def tanh (*self*)

Get the hyperbolic tangent of this instance. Let this instance be x then this method returns $\tanh(x)$.

Parameters:

self

Returns:

The hyperbolic cosine of this instance.

4.37.2.44 def value_of (*value*)

This factory method converts the argument to a complex uncertain value.

Parameters:

value A numeric value.

Returns:

An instance of **CUncertainComponent** (p. 117).

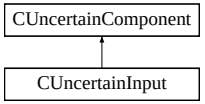
4.37.3 Member Data Documentation

4.37.3.1 __context [private]

4.37.3.2 tuple value_of = staticmethod(value_of) [static]

4.38 CUncertainInput Class Reference

Inheritance diagram for CUncertainInput::



4.38.1 Detailed Description

This class models a complex-valued input of a function.

Public Member Functions

- def **__getstate__**
  This method provides an interface for serializing objects using pickle.
- def **__init__**
  The default constructor.
- def **__setstate__**
  This method provides an interface for deserializing objects using pickle.
- def **depends_on**
  Returns a list that contains this instance only.

- def **get_a_value**
  Get the value as array.
- def **get_dof**
  Get the degrees of freedom assigned to this input.
- def **get_uncertainty**
  If `x == self` *get the uncertainty of the current node, otherwise return a matrix of zeros.*
- def **get_value**
  Get the value of this input.

Private Attributes

- **__avalue**
- **__dof**
- **__jac**
- **__value**

4.38.2 Member Function Documentation

4.38.2.1 def __getstate__ (*self*)

This method provides an interface for serializing objects using pickle.

Parameters:

self

Returns:

The state of this component.

4.38.2.2 def __init__ (*self*, *value*, *u_real*, *u_imag*, *dof* = arithmetic.INFINITY)

The default constructor.

Parameters:

self

value The value of this instance.

u_real The uncertainty of the real part.

u_imag The uncertainty of the imaginary part.

dof The degrees of freedom of the input.

Attention:

You must not declare instances of quantities.QuantityQuantity as uncertain. Instead encapsulate an uncertain value inside a quantity.

See also:

UncertainQuantity.py

4.38.2.3 def __setstate__ (*self*, *state*)

This method provides an interface for deserializing objects using pickle.

Parameters:

self

state The state to be restored.

4.38.2.4 def depends_on (*self*)

Returns a list that contains this instance only.

Parameters:

self

Returns:

A list.

Reimplemented from **CUncertainComponent** (p. 128).

4.38.2.5 def get_a_value (*self*)

Get the value as array.

Parameters:

self

Returns:

The value of this input as array.

Reimplemented from **CUncertainComponent** (p. 128).

4.38.2.6 def get_dof (*self*)

Get the degrees of freedom assigned to this input.

Parameters:

self

Returns:

The degrees of freedom assigned to this input.

4.38.2.7 def get_uncertainty (*self*, *x*)

If `x == self` get the uncertainty of the current node, otherwise return a matrix of zeros.

Parameters:

self

x Another instance of **CUncertainInput** (p. 133)

Returns:

The uncertainty of this instance with respect to the argument.

Reimplemented from **CUncertainComponent** (p. 129).

4.38.2.8 def get_value (*self*)

Get the value of this input.

Parameters:

self

Returns:

The value of this input

Reimplemented from **CUncertainComponent** (p. 129).

4.38.3 Member Data Documentation

4.38.3.1 __avalue `[private]`

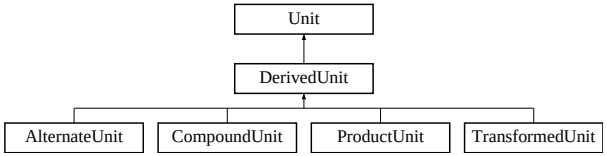
4.38.3.2 __dof `[private]`

4.38.3.3 __jac `[private]`

4.38.3.4 __value [private]

4.39 DerivedUnit Class Reference

Inheritance diagram for DerivedUnit::



4.39.1 Detailed Description

This class provides an abstract interface for all units that have been transformed from other units.

Attention:

This class is intended to be abstract. Instancing it makes no effect.

See also:

AlternateUnit (p. 56)
CompoundUnit (p. 96)
ProductUnit (p. 176)
TransformedUnit (p. 281)

Public Member Functions

- def **__init__**
  abstract default constructor.

4.39.2 Member Function Documentation

4.39.2.1 def __init__ (*self*)

abstract default constructor.

Parameters:

self

4.40 Dimension Class Reference

4.40.1 Detailed Description

This class provides an interface to model physical dimensions.

In order to distinguish between different dimensions, we add an unique symbol to each dimension. In order to aviod confusion, the symbols for physical dimensions must not interfer with the symbols used for base units and alternate units.

See also:

BaseUnit (p. 83)
AlternateUnit (p. 56)

Attention:

This class should not be inherited. It describes the any phyiscal dimension correctly.

Note:

Instances of this class can be serialized using pickle.

Public Member Functions

- def **__div__**
  Return a dimension that describes the fraction of the current and another dimension.
- def **__eq__**
  This function checks if two dimensions are equal.
- def **__getstate__**
  Serialization using pickle.
- def **__init__**
  This is the default constructor.
- def **__mul__**
  Return a dimension that describes the product of the current and another dimension.
- def **__pow__**
  Return a dimension that represents the current dimension raised to an integer power.
- def **__setstate__**
  Deserialization using pickle.
- def **__str__**
  Return a string describing the physical dimension.

- def **get_symbol**
  Same as __eq__.
- def **root**
  Return a dimension that represents the root of the current dimension.

Static Private Attributes

- **__pseudoUnit__** = None

4.40.2 Member Function Documentation

4.40.2.1 def __div__ (*self*, *other*)

Return a dimension that describes the fraction of the current and another dimension.

Parameters:

self
other Another instance of a **Dimension** (p. 138).

Returns:

A new dimension representing the fraction.

4.40.2.2 def __eq__ (*self*, *other*)

This function checks if two dimensions are equal.

Parameters:

self
other Another dimension.

Returns:

True, if the dimensions are equal.

4.40.2.3 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

4.40.2.4 def __init__ (*self*, *value*)

This is the default constructor.

This function creates a physical dimension.

Parameters:

self
value An unique symbol that is used to model the dimension.

Exceptions:

UnitExistsException If the same symbol already exists in the dictionary of units.

4.40.2.5 def __mul__ (*self*, *other*)

Return a dimension that describes the product of the current and another dimension.

Parameters:

self
other Another instance of a **Dimension** (p. 138).

Returns:

A new dimension representing the product.

4.40.2.6 def __pow__ (*self*, *value*)

Return a dimension that represents the current dimension raised to an integer power.

Parameters:

self
value An integer to be used as power.

Returns:

A new dimension representing the power.

4.40.2.7 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self
state The state of the object.

4.40.2.8 def __str__ (*self*)

Return a string describing the physical dimension.

The physical dimensions are described in the same way as units.

Parameters:

self

Returns:

A string describing this dimension.

See also:

Unit.__str__ (p. 320)

4.40.2.9 def get_symbol (*self*)

Same as __eq__.

Parameters:

self

See also:

Dimension.__eq__ (p. 139)

4.40.2.10 def root (*self*, *value*)

Return a dimension that represents the root of the current dimension.

Parameters:

self

value An integer to be used as root.

Returns:

A new dimension representing the root.

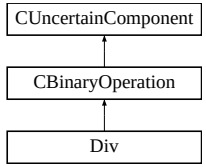
4.40.3 Member Data Documentation

4.40.3.1 __pseudoUnit__ = None `[static, private]`

A pseudo unit representing the physical unit. All Operations that can be performed on units are also applicable to Dimensions. Therefore each dimension is represented internally by a pseudo **Unit** (p. 314).

4.41 Div Class Reference

Inheritance diagram for Div::



4.41.1 Detailed Description

This class models dividing two complex values.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.41.2 Member Function Documentation

4.41.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.41.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

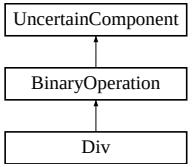
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.42 Div Class Reference

Inheritance diagram for Div::



4.42.1 Detailed Description

This class models GUM-tree nodes that divide two silblings.

Public Member Functions

- def **__init__**
  Default constructor.
- def **arithmetic_check**
  Checks for divide by zero.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = \frac{x_1}{x_2}$ *then the resulting uncertainty is* $u(y) = \frac{u(x_1)}{x_2} - \frac{x_1 \times u(x_2)}{x_2^2}$.
- def **get_value**
  Returns the fraction of the silblings assigned.

4.42.2 Member Function Documentation

4.42.2.1 def __init__ (*self*, *left*, *right*)

Default constructor.

Parameters:

self
left Left silbling of this instance.
right Right silbling of this instance.

Reimplemented from **BinaryOperation** (p. 89).

4.42.2.2 def arithmetic_check (*self*)

Checks for divide by zero.

Parameters:

self

Exceptions:

ArithmeticError If the right silbling returns 0.0 as value.

Reimplemented from **UncertainComponent** (p. 302).

4.42.2.3 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self
other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **BinaryOperation** (p. 89).

4.42.2.4 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = \frac{x_1}{x_2}$ then the resulting uncertainty is $u(y) = \frac{u(x_1)}{x_2} - \frac{x_1 \times u(x_2)}{x_2^2}$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.42.2.5 def get_value (*self*)

Returns the fraction of the silblings assigned.

Parameters:

self

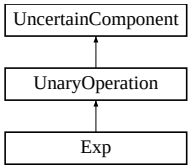
Returns:

A numeric value, representing the fraction of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.43 Exp Class Reference

Inheritance diagram for Exp::



4.43.1 Detailed Description

This class models the GUM-tree-nodes that take the exponential of a silbling.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.

- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = e^x$ then the resulting uncertainty is $u(y) = x \times e^x \times u(x)$.
- def **get_value**
  Returns the exponential of the silbling.

4.43.2 Member Function Documentation

4.43.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self
right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.43.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self
other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.43.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = e^x$ then the resulting uncertainty is $u(y) = x \times e^x \times u(x)$.

Parameters:

self
component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.43.2.4 def get_value (*self*)

Returns the exponential of the silbling.

Parameters:

self

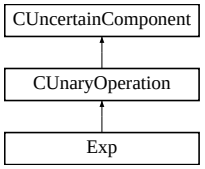
Returns:

A numeric value, representing the exponential of the silbling.

Reimplemented from **UncertainComponent** (p. 305).

4.44 Exp Class Reference

Inheritance diagram for Exp::



4.44.1 Detailed Description

This class models the exponential function e^x. x denotes the sibling of this instance.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.44.2 Member Function Documentation

4.44.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.44.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

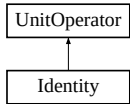
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.45 Identity Class Reference

Inheritance diagram for Identity::



4.45.1 Detailed Description

This class provides an Interface for the identity Operator.

This class returns all values as are.

Attention:

This class is intendend to be static final. Deriving subclasses makes no sense since there is only one identity operator. Also Instances of this class should be avoided. If you need an identity operator reference it from the global IDENTITY object of this module.

Note:

Instances of this class can be serialized using pickle.

Public Member Functions

- def **__eq__**
  Test for equality.
- def **__getstate__**
  Serialization using pickle.
- def **__invert__**
  Invert the current operation.
- def **__mul__**
  Perform the current operation on another operator.
- def **__setstate__**
  Deserialization using pickle.
- def **__str__**
  Represent this operation by a string.
- def **convert**
  Convert a value.
- def **is_linear**
  Check if the Operator is linear.

4.45.2 Member Function Documentation

4.45.2.1 def __eq__ (*self*, *other*)

Test for equality.

Parameters:

self
other Another **UnitOperator** (p. 326).

Reimplemented from **UnitOperator** (p. 327).

4.45.2.2 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **UnitOperator** (p. 327).

4.45.2.3 def __invert__ (*self*)

Invert the current operation.

This method returns the inverse Operation of the current operation.

Parameters:

self

Returns:

The inverse Operation of the current Operation.

Warning:

This method is intended to be abstract. You have to override it in order to get any effect.

Reimplemented from **UnitOperator** (p. 327).

4.45.2.4 def __mul__ (*self*, *otherOperator*)

Perform the current operation on another operator.

This method returns the parameter.

Parameters:

self
otherOperator The other operator to concat.

Returns:

The other Operator.

Reimplemented from **UnitOperator** (p. 328).

4.45.2.5 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self
state The state of the object.

Reimplemented from **UnitOperator** (p. 328).

4.45.2.6 def __str__ (*self*)

Represent this operation by a string.

Parameters:

self

Returns:

A string describing this operation.

Reimplemented from **UnitOperator** (p. 328).

4.45.2.7 def convert (*self*, *value*)

Convert a value.

This method returns the parameter.

Parameters:

self

value The value to convert (will be returned).

Returns:

The parameter value

Reimplemented from **UnitOperator** (p. 329).

4.45.2.8 def is_linear (*self*)

Check if the Operator is linear.

Identity (p. 148) is a linear operator. Thus, this method always returns True.

Parameters:

self

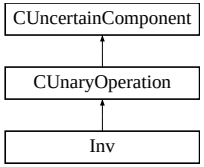
Returns:

`True`.

Reimplemented from **UnitOperator** (p. 329).

4.46 Inv Class Reference

Inheritance diagram for Inv::



4.46.1 Detailed Description

This class models inverting complex values. Let an instance of this class model the complex value x then this class models $\frac{1}{x}$.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.46.2 Member Function Documentation

4.46.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.46.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

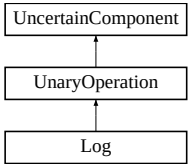
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.47 Log Class Reference

Inheritance diagram for Log::



4.47.1 Detailed Description

This class models the GUM-tree-nodes that take the Natural Logarithm of a silbling.

Public Member Functions

- def **__init__**
  Default constructor.
- def **arithmetic_check**
  Checks for undefined arguments.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = ln(x)$ then the resulting uncertainty is $u(y) = \frac{1}{x}u(x)$.
- def **get_value**
  Returns the Natural Logarithm of the silbling.

4.47.2 Member Function Documentation

4.47.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self
right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.47.2.2 def arithmetic_check (*self*)

Checks for undefined arguments.

Note:

The natural logarithm is not defined for values $x \leq 0$.

Parameters:

self

Exceptions:

ArithmeticError If $x \leq 0$.

Reimplemented from **UncertainComponent** (p. 302).

4.47.2.3 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self
other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.47.2.4 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = ln(x)$ then the resulting uncertainty is $u(y) = \frac{1}{x}u(x)$.

Parameters:

self
component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.47.2.5 def get_value (*self*)

Returns the Natural Logarithm of the silbling.

Parameters:

self

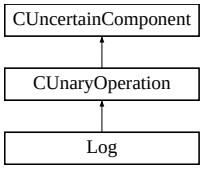
Returns:

A numeric value, representing the Natural Logarithm of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.48 Log Class Reference

Inheritance diagram for Log::



4.48.1 Detailed Description

This class models logarithms having a real base. However, the base cannot be uncertain.

Public Member Functions

- def **__init__**
  The default constructor.
- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

Private Attributes

- **__base**

4.48.2 Member Function Documentation

4.48.2.1 def __init__ (*self*, *sibling*, *base* = `numpy.e`)

The default constructor.

Parameters:

self
sibling The sibling of this instance.
base The base of the logarithm (must be a real number).

4.48.2.2 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.48.2.3 def get_value (*self*)

Get the value of this component.

Parameters:

self

Returns:

The value of this component.

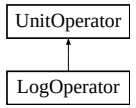
Reimplemented from **CUncertainComponent** (p. 129).

4.48.3 Member Data Documentation

4.48.3.1 __base `[private]`

4.49 LogOperator Class Reference

Inheritance diagram for LogOperator::



4.49.1 Detailed Description

This class provides an interface for logarithmic operators.

Note:

Instances of this class can be serialized using pickle.

Public Member Functions

- def **__eq__**
  Test for equality.
- def **__getstate__**
  Serialization using pickle.
- def **__init__**
  Default constructor.
- def **__invert__**
  Invert the current operation.
- def **__setstate__**
  Deserialization using pickle.
- def **__str__**
  Represent this operation by a string.
- def **convert**
  Convert a value.
- def **get_base**
  Get the base of this logarithm.
- def **is_linear**
  Check if this operator is linear.

Private Attributes

- **__base__**
- **__logBase__**

4.49.2 Member Function Documentation

4.49.2.1 def __eq__ (*self*, *other*)

Test for equality.

Parameters:

self
other Another **UnitOperator** (p. 326).

Reimplemented from **UnitOperator** (p. 327).

4.49.2.2 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **UnitOperator** (p. 327).

4.49.2.3 def __init__ (*self*, *base*)

Default constructor.

Initializes this operator and assigns the base to it.

Parameters:

self
base The base of the logarithm.

4.49.2.4 def __invert__ (*self*)

Invert the current operation.

This method returns the inverse Operation of the current operation.

Parameters:

self The current instance of this class.

Returns:

The inverse Operation of the current Operation.

Reimplemented from **UnitOperator** (p. 327).

4.49.2.5 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

Reimplemented from **UnitOperator** (p. 328).

4.49.2.6 def __str__ (*self*)

Represent this operation by a string.

Parameters:

self

Returns:

A string describing this operation.

Reimplemented from **UnitOperator** (p. 328).

4.49.2.7 def convert (*self*, *value*)

Convert a value.

This method performs the logarithm on an absolute value.

Attention:

The logarithm for complex values is not defined.

Parameters:

self

value The value to convert.

Exceptions:

TypeError If the argument is a complex number.

Returns:

The converted value

Reimplemented from **UnitOperator** (p. 329).

4.49.2.8 def get_base (*self*)

Get the base of this logarithm.

This method returns the base of the logarithm.

Parameters:

self

Returns:

The base of the logarithm

4.49.2.9 def is_linear (*self*)

Check if this operator is linear.

This operator is not linear.

Parameters:

self

Returns:

False

Reimplemented from **UnitOperator** (p. 329).

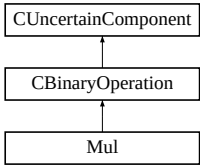
4.49.3 Member Data Documentation

4.49.3.1 __base__ [private]

4.49.3.2 __logBase__ [private]

4.50 Mul Class Reference

Inheritance diagram for Mul::



4.50.1 Detailed Description

This class models multiplying two complex values.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.50.2 Member Function Documentation

4.50.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.50.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

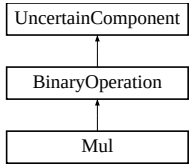
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.51 Mul Class Reference

Inheritance diagram for Mul::



4.51.1 Detailed Description

This class models GUM-tree nodes that multiply two silblings.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = x_1 \times x_2$ then the resulting uncertainty is $u(y) = x_2 \times u(x_1) + x_1 \times u(x_2)$.
- def **get_value**
  Returns the product of the silblings assigned.

4.51.2 Member Function Documentation

4.51.2.1 def __init__ (*self*, *left*, *right*)

Default constructor.

Parameters:

self

left Left silbling of this instance.

right Right silbling of this instance.

Reimplemented from **BinaryOperation** (p. 89).

4.51.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **BinaryOperation** (p. 89).

4.51.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = x_1 \times x_2$ then the resulting uncertainty is $u(y) = x_2 \times u(x_1) + x_1 \times u(x_2)$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.51.2.4 def get_value (*self*)

Returns the product of the silblings assigned.

Parameters:

self

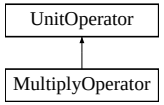
Returns:

A numeric value, representing the product of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.52 MultiplyOperator Class Reference

Inheritance diagram for MultiplyOperator::



4.52.1 Detailed Description

This class provides an Interface for factor operators.

This class multiplies a constant with an existing Operator.

Note:

Instances of this class can be serialized using pickle.

Public Member Functions

- def **__eq__**
  Test for equality.
- def **__getstate__**
  Serialization using pickle.
- def **__init__**
  Default constructor.
- def **__invert__**
  Invert the current operation.
- def **__mul__**
  Perform the current operation on another operator.
- def **__setstate__**
  Deserialization using pickle.
- def **__str__**
  Represent this operation by a string.
- def **convert**
  Convert a value.
- def **get_factor**
  Get the factor.
- def **is_linear**
  Check if the operator is linear.

Private Member Functions

- def **__isNegative**
  Helper method to optimize comparsions.

Private Attributes

- **__factor__**

4.52.2 Member Function Documentation

4.52.2.1 def __eq__ (*self*, *other*)

Test for equality.

Parameters:

self

other Another **UnitOperator** (p. 326).

Reimplemented from **UnitOperator** (p. 327).

4.52.2.2 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **UnitOperator** (p. 327).

4.52.2.3 def __init__ (*self*, *factor*)

Default constructor.

Initializes this operator and assigns the factor to the current operator.

Parameters:

self

factor The offset of this operator.

4.52.2.4 def __invert__ (*self*)

Invert the current operation.

For example let this operator be $a \times f(x)$ then the inverse is $\frac{1}{a} \times f(x)$.

Parameters:

self

Returns:

The inverse Operation of the current Operation.

Reimplemented from **UnitOperator** (p. 327).

4.52.2.5 def __isNegative (*positvieOp*, *negativeOp*) [private]

Helper method to optimize comparsions.

Parameters:

negativeOp An **MultiplyOperator** (p. 163).

positvieOp An **MultiplyOperator** (p. 163).

Returns:

`negativeOp.get_factor() == ~positvieOp.get_factor()` (p. 167)

4.52.2.6 def __mul__ (*self*, *otherOperator*)

Perform the current operation on another operator.

The current operation (muliplying by a) will be performed on another operator $f(x)$. So that the new operator is $a \times g(x)$. In order to avoid numerical quirks, this method checks wether the parameter is an instance of a MuliplyOperator. If yes, then only the factor is updated.

Parameters:

self

otherOperator The other operator to concat.

Returns:

The resulting operator.

Reimplemented from **UnitOperator** (p. 328).

4.52.2.7 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

Reimplemented from **UnitOperator** (p. 328).

4.52.2.8 def __str__ (*self*)

Represent this operation by a string.

Parameters:

self

Returns:

A string describing this operation.

Reimplemented from **UnitOperator** (p. 328).

4.52.2.9 def convert (*self*, *value*)

Convert a value.

This method performs the multiplication with an factor on an absolute value.

Parameters:

self

value The value to convert.

Returns:

The converted value.

Reimplemented from **UnitOperator** (p. 329).

4.52.2.10 def get_factor (*self*)

Get the factor.

This method returns the factor of this operator.

Parameters:

self

Returns:

The factor of this operator.

4.52.2.11 def is_linear (*self*)

Check if the operator is linear.

This operator is linear.

Parameters:

self

Returns:

True

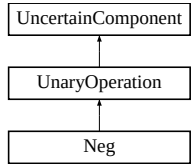
Reimplemented from **UnitOperator** (p. 329).

4.52.3 Member Data Documentation

4.52.3.1 __factor__ [private]

4.53 Neg Class Reference

Inheritance diagram for Neg::



4.53.1 Detailed Description

This class models the unary negation as GUM-tree-element.

Public Member Functions

- def **__init__**

  Default constructor.

- def **equal_debug**

  A method that is only used for serialization checking.

- def **get_uncertainty**

  Returns the uncertainty of this node. Let the node represent the operation $y = -x$ then the resulting uncertainty is $u(y) = -u(x)$.

- def **get_value**

  Returns the exponential of the silbling.

4.53.2 Member Function Documentation

4.53.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self

right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.53.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.53.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = -x$ then the resulting uncertainty is $u(y) = -u(x)$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.53.2.4 def get_value (*self*)

Returns the exponential of the silbling.

Parameters:

self

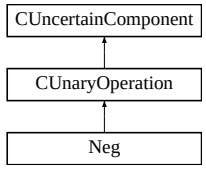
Returns:

A numeric value, representing the negative value of the silbling.

Reimplemented from **UncertainComponent** (p. 305).

4.54 Neg Class Reference

Inheritance diagram for Neg::



4.54.1 Detailed Description

This class models taking the negative of a complex value.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.54.2 Member Function Documentation

4.54.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.54.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

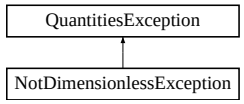
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.55 NotDimensionlessException Class Reference

Inheritance diagram for NotDimensionlessException::



4.55.1 Detailed Description

Exception that is raised whenever a a unit is not dimensionless where it has to be.

Public Member Functions

- def **__init__**
  Default constructor.
- def **__str__**
  Returns a string describing this exception.

Private Attributes

- **__unit__**

4.55.2 Member Function Documentation

4.55.2.1 def __init__ (*self*, *unit*, *args*)

Default constructor.

Parameters:

self

unit Instance of a unit that raised the exception.

args Additional arguments of this exception.

4.55.2.2 def __str__ (*self*)

Returns a string describing this exception.

Parameters:

self

Returns:

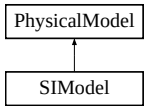
A string that describes the exception.

4.55.3 Member Data Documentation

4.55.3.1 __unit__ [private]

4.56 PhysicalModel Class Reference

Inheritance diagram for PhysicalModel::



4.56.1 Detailed Description

This class models the abstract interface for physical models.

This class provides an interface for defining physical models. Up to now, the subclasses do not model relativistic effects. Therefore this interface might be extended in the next version.

Attention:

This class is only an abstract interface. You will have to override it in order to get any effect.

See also:

si.SIModel (p. 231)

Public Member Functions

- def **__init__**
  This is the default constructor.
- def **get_dimension**
  Get the pysical dimension that corresponds to the given unit.

4.56.2 Member Function Documentation

4.56.2.1 def __init__ (*self*)

This is the default constructor.

Parameters:

self

Reimplemented in **SIModel** (p. 232).

4.56.2.2 def get_dimension (*self*, *unit*)

Get the pysical dimension that corresponds to the given unit.

Parameters:

self
unit to check the dimension for.

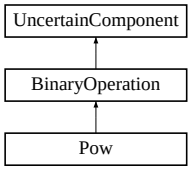
Returns:

The corresponding physical dimension.

Reimplemented in **SIModel** (p. 232).

4.57 Pow Class Reference

Inheritance diagram for Pow::



4.57.1 Detailed Description

This class models GUM-tree nodes that raise the left silbling to the power of the right one.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = x_1^{x_2}$ *then the resulting uncertainty is* $u(y) = x_2 \times x_1^{x_2-1} \times u(x_1) + x_1^{x_2} \times ln(x_1) \times u(x_2)$.
- def **get_value**
  Returns the power $pow(left, right)$ *of the silblings assigned.*

4.57.2 Member Function Documentation

4.57.2.1 def __init__ (*self*, *left*, *right*)

Default constructor.

Parameters:

self
left Left silbling of this instance.
right Right silbling of this instance.

Reimplemented from **BinaryOperation** (p. 89).

4.57.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self
other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **BinaryOperation** (p. 89).

4.57.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = x_1^{x_2}$ then the resulting uncertainty is $u(y) = x_2 \times x_1^{x_2-1} \times u(x_1) + x_1^{x_2} \times ln(x_1) \times u(x_2)$.

Attention:

The uncertainty is only defined, if $x_1 > 0$ and $x_1 > 0$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Exceptions:

ArithmeticError If $x_1 \leq 0$ or $x_2 \leq 0$.

Reimplemented from **UncertainComponent** (p. 304).

4.57.2.4 def get_value (*self*)

Returns the power $pow(left, right)$ of the silblings assigned.

Parameters:

self

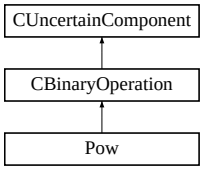
Returns:

A numeric value, representing the power of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.58 Pow Class Reference

Inheritance diagram for Pow::



4.58.1 Detailed Description

This class models complex powers.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.58.2 Member Function Documentation

4.58.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.58.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

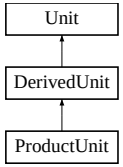
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.59 ProductUnit Class Reference

Inheritance diagram for ProductUnit::



4.59.1 Detailed Description

The unit is a combined unit of the product of the powers of units.

The unit is stored in its canonical form. That is the simplest form. For example $[m] := \left[\frac{m^2}{m}\right]$.

Note:

Instances of this class can be serialized using pickle.

Public Member Functions

- def **__div__**
  Divide two units.
- def **__eq__**
  Checks if two product units are equal.
- def **__getstate__**
  Serialization using pickle.
- def **__init__**
  Default constructor.
- def **__setstate__**
  Deserialization using pickle.
- def **__str__**
  Print the current unit. This function returns a string of the form `factor1*factor2`.
- def **get_system_unit**
  Get the corresponding system unit.
- def **get_unit**
  Returns the unit at the given index.

- def **get_unitCount**
  Get the total count of factors of this product unit.
- def **get_unitPow**
  Get the power exponent of a factor at the given index.
- def **get_unitRoot**
  Get the root exponent of a factor at the given index.
- def **normalize**
  This function merge duplicate factors and converts this unit into its canonical form.
- def **strip_unit**
  Return the contained unit of a unit, if it is a product unit, contains only one element, and has an exponent equal to one.
- def **to_system_unit**
  Get the operator to convert to the system unit. This method concatenates the individual operators and returns the joint operator to the system unit, if this unit is not a system unit.
- def **value_of**
  Factory method for generating product units. Used to compare other units.

Static Public Attributes

- tuple **strip_unit** = staticmethod(**strip_unit**)
- tuple **value_of** = staticmethod(**value_of**)

Private Member Functions

- def **__cloneElements**
  Return a copy of the sequence of factors.
- def **__isSystemUnit**
  Check if the current unit is a system unit. This product unit is a system unit, if all of its factors are system units.

Private Attributes

- **__elements__**

Static Private Attributes

- list **__elements__** = [ ]

4.59.2 Member Function Documentation

4.59.2.1 def __cloneElements (*self*) [private]

Return a copy of the sequence of factors.

Parameters:

self

Returns:

A copy of __elements__.

See also:

self.__elements__

4.59.2.2 def __div__ (*self*, *other*)

Divide two units.

Parameters:

self

other A divisor.

See also:

Unit.__div__ (p. 317)

Reimplemented from **Unit** (p. 317).

4.59.2.3 def __eq__ (*self*, *other*)

Checks if two product units are equal.

Parameters:

self

other **Unit** (p. 314) to compare to

Returns:

True If the units are equal, False if the units are unequal.

Reimplemented from **Unit** (p. 317).

4.59.2.4 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **Unit** (p. 317).

4.59.2.5 def __init__ (*self*, *left* = None, *right* = None)

Default constructor.

Parameters:

self

left A unit to left-multiply.

right A unit to right-multiply.

4.59.2.6 def __isSystemUnit (*self*) [private]

Check if the current unit is a system unit. This product unit is a system unit, if all of its factors are system units.

Parameters:

self

Returns:

True if it is a system unit.

4.59.2.7 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

Reimplemented from **Unit** (p. 320).

4.59.2.8 def __str__ (*self*)

Print the current unit. This function returns a string of the form `factor1*factor2`.

Parameters:

self

Returns:

A string describing this unit.

See also:

__ProductElement.__str__

Reimplemented from **Unit** (p. 320).

4.59.2.9 def get_system_unit (*self*)

Get the corresponding system unit.

If no system unit is found, the unit is formed from the system units of the factors of the current unit.

Returns:

The corresponding system unit.

Reimplemented from **Unit** (p. 322).

4.59.2.10 def get_unit (*self*, *index*)

Returns the unit at the given index.

Parameters:

self
index Index of the desired unit.

Returns:

The unit at index.

4.59.2.11 def get_unitCount (*self*)

Get the total count of factors of this product unit.

Parameters:

self

Returns:

The number of factors.

4.59.2.12 def get_unitPow (*self*, *index*)

Get the power exponent of a factor at the given index.

Attention:

Since roots are rational, you have to call `get_unitPow` and `get_unitRoot` in order to obtain the complete exponent of this unit. For example for $\sqrt{m^3}$ the results would be 3 for `get_unitPow` and 2 for `get_unitRoot`.

Parameters:

self
index Index of the desired unit.

Returns:

The (integer) power of the current unit

4.59.2.13 def get_unitRoot (*self*, *index*)

Get the root exponent of a factor at the given index.

Attention:

Since roots are rational, you have to call `get_unitPow` and `get_unitRoot` in order to obtain the complete exponent of this unit. For example for $\sqrt{m^3}$ the results would be 3 for `get_unitPow` and 2 for `get_unitRoot`.

Parameters:

self
index Index of the desired unit.

Returns:

The (integer) root of the current unit

4.59.2.14 def normalize (*self*)

This function merge duplicate factors and converts this unit into its canonical form.

Parameters:

self

4.59.2.15 def strip_unit (*unit*)

Return the contained unit of a unit, if it is a product unit, contains only one element, and has an exponent equal to one.

4.59.2.16 def to_system_unit (*self*)

Get the operator to convert to the system unit. This method concatenates the individual operators and returns the joint operator to the system unit, if this unit is not a system unit.

Returns:

The operator to the system unit.

Exceptions:

qexceptions.ConversionException *(*p. *108)* If one of the system units is formed using a non linear operator, or if a factor has a rational exponent.

Reimplemented from **Unit** (p. 324).

4.59.2.17 def value_of (*unit*)

Factory method for generating product units. Used to compare other units.

Parameters:

unit A unit.

Returns:

The argument, if it is a product unit, or a new instance of **ProductUnit** (p. 176) if the argument is not a product unit.

4.59.3 Member Data Documentation

4.59.3.1 __elements__ [private]

4.59.3.2 list __elements__ = [ ] [static, private]

The factors forming the product unit.

See also:

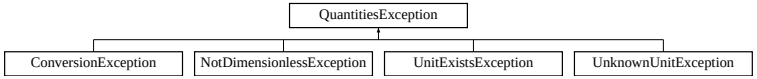
__ProductElement__ (p. 41)

4.59.3.3 tuple strip_unit = staticmethod(strip_unit) [static]

4.59.3.4 tuple value_of = staticmethod(value_of) [static]

4.60 QuantitiesException Class Reference

Inheritance diagram for QuantitiesException::



4.60.1 Detailed Description

General class for qexceptions of this module.

Public Member Functions

- def **__init__**
  Default constructor.

4.60.2 Member Function Documentation

4.60.2.1 def __init__ (*self*, *args*)

Default constructor.

Parameters:

self

args Arguments of this exception

4.61 Quantity Class Reference

4.61.1 Detailed Description

Base class that provides an interface to model quantities.

Note:

The numeric types (i.e. int, float, long, complex, and **arithmetic.RationalNumber** (p. 210)) are automatically transformed to an dimensionless quantity if the operations are performed on them. This also applies if a quantity is the right operand of the numeric types stated above.
Instances of this class can be serialized using pickle.

Public Member Functions

- def **__abs__**

  *Get the absolute value of this **Quantity*** (p. *184).*

- def **__add__**

  *Get the sum of another instance of **Quantity*** (p. *184) and this instance.*

- def **__cmp__**

  Compare two instances of quantity.

- def **__coerce__**

  Implementation of coercion rules.

- def **__complex__**

  Cast this instance to the numeric type complex.

- def **__div__**

  *Get the fraction of another instance of **Quantity*** (p. *184) and this instance.*

- def **__eq__**

  Check, if this instance is equal to the argument. A comparsion will be done, if the units are comparable.

- def **__float__**

  Cast this instance to the numeric type float.

- def **__ge__**

  Check, if this instance is greater or equal to the argument. A comparsion will be done, if the units are comparable.

- def **__getstate__**

  Serialization using pickle.

- def **__gt__**

  Check, if this instance is greater than the argument. A comparsion will be done, if the units are comparable.

- def **__iadd__**

  Add the argument to this instance.

- def **__idiv__**

  Divide this instance by the argument.

- def **__imul__**

  Multiply this instance with the argument.

- def **__init__**

  Default constructor.

- def **__int__**

  Cast this instance to the numeric type int.

- def **__invert__**

  *Return the inverted instance of this **Quantity*** (p. *184). For example, let your quantity be* $\frac{1}{2}\frac{m}{s}$*, then the result of this operation is* $2\frac{s}{m}$*.*

- def **__ipow__**

  Raise the this instance to the argument.

- def **__isub__**

  Substract the argument from this instance.

- def **__le__**

  Check, if this instance is less or equal to the argument. A comparsion will be done, if the units are comparable.

- def **__long__**

  Cast this instance to the numeric type long.

- def **__lt__**

  Check, if this instance is less than the argument. A comparsion will be done, if the units are comparable.

- def **__mul__**

  *Get the product of another instance of **Quantity*** (p. *184) and this instance.*

- def **__ne__**

  Check, if this instance is not equal to the argument. A comparsion will be done, if the units are comparable.

- def **__neg__**

  Negate the value of this quantity.

- def **__pos__**

  Copy this instance.

- def **__pow__**

  Get the power of of this instance.

- def **__radd__**

  *Get the sum of this instance of **Quantity*** (p. *184) and another value.*

- def **__rdiv__**

Get the fraction of another value and this instance.

- def **__rmul__**

  *Get the product of this instance of **Quantity** (p. 184) and another value.*

- def **__rpow__**

  Get the power of another value and this instance.

- def **__rsub__**

  *Get the difference of another value and this instance of **Quantity** (p. 184).*

- def **__setstate__**

  Deserialization using pickle.

- def **__str__**

  *Get a string describing this **Quantity** (p. 184). The result will be of the form* `value unit` *(i.e. "12.0 m").*

- def **__sub__**

  *Get the difference of another instance of **Quantity** (p. 184) and this instance.*

- def **arccos**

  This method provides the broadcast interface for numpy.arccos.

- def **arccosh**

  This method provides the broadcast interface for numpy.arccosh.

- def **arcsin**

  This method provides the broadcast interface for numpy.arcsin.

- def **arcsinh**

  This method provides the broadcast interface for numpy.arcsinh.

- def **arctan**

  This method provides the broadcast interface for numpy.arctan.

- def **arctan2**

  This method provides the broadcast interface for numpy.arctan2.

- def **arctanh**

  This method provides the broadcast interface for numpy.arctanh.

- def **ceil**

  This method provides the broadcast interface for numpy.ceil.

- def **conjugate**

This method provides the broadcast interface for numpy.conjugate.

- def **cos**

  This method provides the broadcast interface for numpy.cos.

- def **cosh**

  This method provides the broadcast interface for numpy.cosh.

- def **exp**

  This method provides the broadcast interface for numpy.exp.

- def **fabs**

  This method provides the broadcast interface for numpy.fabs.

- def **floor**

  This method provides the broadcast interface for numpy.floor.

- def **get_default_unit**

  Get the unit that is used commonly for this quantity.

- def **get_value**

  Get the absolute value of the quantity using the specified unit.

- def **hypot**

  This method provides the broadcast interface for numpy.arctan2.

- def **is_dimensionless**

  Check if this quantity is dimensionless.

- def **is_strict**

  Get the type of quantities calculation. If either strict or non strict evaluation of quantities is implemented.

- def **log**

  This method provides the broadcast interface for numpy.log.

- def **log10**

  This method provides the broadcast interface for numpy.log10.

- def **log2**

  This method provides the broadcast interface for numpy.log2.

- def **set_strict**

  Turn on/off the strict evaluation of quantities. This will affect any quantities calculation beyond this point.

- def **sin**

This method provides the broadcast interface for numpy.sin.

- def **sinh**
  This method provides the broadcast interface for numpy.sinh.
- def **sqrt**
  This method provides the broadcast interface for numpy.sqrt.
- def **square**
  This method provides the broadcast interface for numpy.sqrt.
- def **tan**
  This method provides the broadcast interface for numpy.tan.
- def **tanh**
  This method provides the broadcast interface for numpy.tanh.
- def **value_of**
  Factory for generating quantities.

Static Public Attributes

- tuple **is_strict** = staticmethod(**is_strict**)
- tuple **set_strict** = staticmethod(**set_strict**)
- tuple **value_of** = staticmethod(**value_of**)

Private Member Functions

- def **__accuracy**
  Helper method, to increase the accuracy of integer operations. As soon an int or long is provided, it is converted to a rational number.
- def **__unitComparsion**
  Helper method.

Private Attributes

- **__unit__**
- **__value__**

Static Private Attributes

- tuple **__accuracy** = staticmethod(**__accuracy**)
- **__STRICT** = True
- tuple **__unitComparsion** = staticmethod(**__unitComparsion**)

4.61.2 Member Function Documentation

4.61.2.1 def __abs__ (*self*)

Get the absolute value of this **Quantity** (p. 184).

Parameters:

self

Returns:

The absolute value of this quantity.

4.61.2.2 def __accuracy (*value*) [private]

Helper method, to increase the accuracy of integer operations. As soon an int or long is provided, it is converted to a rational number.

Parameters:

value The value to be converted.

4.61.2.3 def __add__ (*self*, *other*)

Get the sum of another instance of **Quantity** (p. 184) and this instance.

Parameters:

self

other Another instance of **Quantity** (p. 184) or numeric value.

Returns:

A new instance of **Quantity** (p. 184) representing the sum of both quantities.

Exceptions:

qexceptions.ConversionException *(*p. ***108)*** If the units are not comparable.

4.61.2.4 def __cmp__ (*self*, *other*)

Compare two instances of quantity.

Parameters:

self

other Another instance of **Quantity** (p. 184) or numeric value.

Returns:

-1, if the this instance is less than the argument, 0, if this instance is equal to the argument, +1, if this instance is greater than the argument.

Exceptions:

qexceptions.ConversionException *(*p. ***108)*** If the units are not comparable.

4.61.2.5 def __coerce__ (*self*, *other*)

Implementation of coercion rules.

See also:

Coercion - The page describing the coercion rules.

4.61.2.6 def __complex__ (*self*)

Cast this instance to the numeric type complex.

Attention:

All information about the unit used will be stripped from the result.

Parameters:

self

Returns:

The value of this instance casted to complex.

4.61.2.7 def __div__ (*self*, *other*)

Get the fraction of another instance of **Quantity** (p. 184) and this instance.

Attention:

This method performs no conversion of alternate units: Even if the units are defined in the same dimension. For example, if one takes $m \div ft$ the result will be $\frac{ft}{m}$ not dimensionless.

Parameters:

self The dividend.

other Another instance of **Quantity** (p. 184) or numeric value used as divisor.

Returns:

A new instance of **Quantity** (p. 184) representing the sum of both quantities.

4.61.2.8 def __eq__ (*self*, *other*)

Check, if this instance is equal to the argument. A comparsion will be done, if the units are comparable.

Parameters:

self

other Another instance of **Quantity** (p. 184).

Returns:

True, if this instance is equal to the argument.

4.61.2.9 def __float__ (*self*)

Cast this instance to the numeric type float.

Attention:

All information about the unit used will be stripped from the result.
This conversion is only possible, if weak consitency checking is enabled.

Parameters:

self

Returns:

The value of this instance casted to float.

4.61.2.10 def __ge__ (*self*, *other*)

Check, if this instance is greater or equal to the argument. A comparsion will be done, if the units are comparable.

Parameters:

self

other Another instance of **Quantity** (p. 184).

Returns:

True, if this instance is greater or equal to the argument.

Exceptions:

qexceptions.ConversionException *(*p. ***108)*** If the units are not comparable.

4.61.2.11 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

4.61.2.12 def __gt__ (*self*, *other*)

Check, if this instance is greater than the argument. A comparsion will be done, if the units are comparable.

Parameters:

self
other Another instance of **Quantity** (p. 184).

Returns:

True, if this instance is greater than the argument.

Exceptions:

qexceptions.ConversionException *(*p. ***108****)* If the units are not comparable.

4.61.2.13 def __iadd__ (*self*, *other*)

Add the argument to this instance.

Parameters:

self
other Another instance of **Quantity** (p. 184) or numeric value.

Exceptions:

qexceptions.ConversionException *(*p. ***108****)* If the units are not comparable.

4.61.2.14 def __idiv__ (*self*, *other*)

Divide this instance by the argument.

Parameters:

self
other Another instance of **Quantity** (p. 184) or numeric value..

4.61.2.15 def __imul__ (*self*, *other*)

Multiply this instance with the argument.

Parameters:

self
other Another instance of **Quantity** (p. 184) or numeric value.

4.61.2.16 def __init__ (*self*, *unit*, *value*)

Default constructor.

Parameters:

self The current instance of this class.
unit The corresponding unit.
value The value assigned

Note:

You may use numeric values, sequence types, and instances of **ucomponents.UncertainInput** (p. 309) as values.

See also:

units.Unit (p. 314)
units.Dimensions

4.61.2.17 def __int__ (*self*)

Cast this instance to the numeric type int.

Attention:

All information about the unit used will be stripped from the result.

Parameters:

self

Returns:

The value of this instance casted to int.

4.61.2.18 def __invert__ (*self*)

Return the inverted instance of this **Quantity** (p. 184). For example, let your quantity be $\frac{1}{2}\frac{m}{s}$, then the result of this operation is $2\frac{s}{m}$.

Parameters:

self

Returns:

The inverted quantity.

4.61.2.19 def __ipow__ (*self*, *other*)

Raise the this instance to the argument.

Parameters:

self

other Another instance of **Quantity** (p. 184) or numeric value.

4.61.2.20 def __isub__ (*self*, *other*)

Substract the argument from this instance.

Parameters:

self

other Another instance of **Quantity** (p. 184) or numeric value.

Exceptions:

qexceptions.ConversionException *(*p. ***108)*** If the units are not comparable.

4.61.2.21 def __le__ (*self*, *other*)

Check, if this instance is less or equal to the argument. A comparsion will be done, if the units are comparable.

Parameters:

self

other Another instance of **Quantity** (p. 184).

Returns:

True, if this instance is less or equal to the argument.

Exceptions:

qexceptions.ConversionException *(*p. ***108)*** If the units are not comparable.

4.61.2.22 def __long__ (*self*)

Cast this instance to the numeric type long.

Attention:

All information about the unit used will be stripped from the result.

Parameters:

self

Returns:

The value of this instance casted to long.

4.61.2.23 def __lt__ (*self*, *other*)

Check, if this instance is less than the argument. A comparsion will be done, if the units are comparable.

Parameters:

self

other Another instance of **Quantity** (p. 184).

Returns:

True, if this instance is less than the argument.

Exceptions:

qexceptions.ConversionException *(*p. ***108)*** If the units are not comparable.

4.61.2.24 def __mul__ (*self*, *other*)

Get the product of another instance of **Quantity** (p. 184) and this instance.

Attention:

This method performs no conversion of alternate units: Even if the units are defined in the same dimension. For example, if one takes $m \times ft$ the result will be ftm not m^2 nor ft^2.

Parameters:

self

other Another instance of **Quantity** (p. 184) or numeric value.

Returns:

A new instance of **Quantity** (p. 184) representing the product of both quantities.

4.61.2.25 def __ne__ (*self*, *other*)

Check, if this instance is not equal to the argument. A comparsion will be done, if the units are comparable.

Parameters:

self

other Another instance of **Quantity** (p. 184).

Returns:

True, if this instance is not equal to the argument.

4.61.2.26 def __neg__ (*self*)

Negate the value of this quantity.

Parameters:

self

Returns:

A new instance of **Quantity** (p. 184) representing the negative of this quantity.

4.61.2.27 def __pos__ (*self*)

Copy this instance.

Parameters:

self

Returns:

A copy of the current instance.

4.61.2.28 def __pow__ (*self*, *other*)

Get the power of of this instance.

Parameters:

self

other The power to which this instance is raised (must be an integer or dimensionless quantity).

Returns:

A new instance of **Quantity** (p. 184) representing the power of this instance.

Exceptions:

qexceptions.ConversionException *(p. 108)* If a quantity argument is not dimensionless.

See also:

units.Unit.__pow__ (p. 319)

4.61.2.29 def __radd__ (*self*, *other*)

Get the sum of this instance of **Quantity** (p. 184) and another value.

Attention:

This library assumes that this is a commutative operation.

Parameters:

self

other Another value (not an instance of **Quantity** (p. 184)).

Returns:

A new instance of **Quantity** (p. 184) representing the sum.

Exceptions:

qexceptions.ConversionException *(p. 108)* If the units are not comparable.

4.61.2.30 def __rdiv__ (*self*, *other*)

Get the fraction of another value and this instance.

Parameters:

self The divisor.

other Another instance of **Quantity** (p. 184) or numeric value used as dividend.

Returns:

A new instance of **Quantity** (p. 184) representing the sum of both quantities.

4.61.2.31 def __rmul__ (*self*, *other*)

Get the product of this instance of **Quantity** (p. 184) and another value.

Attention:

This library assumes that this is a commutative operation.

Parameters:

self

other Another value (not an instance of **Quantity** (p. 184)).

Returns:

A new instance of **Quantity** (p. 184) representing the product of both quantities.

4.61.2.32 def __rpow__ (*self*, *other*)

Get the power of another value and this instance.

Attention:

In contrast to **Quantity.__pow__** (p. 197) this instance also accepts floating point powers.

Parameters:

self

other Another instance of **Quantity** (p. 184).

Returns:

A new instance of **Quantity** (p. 184) representing the sum of both quantities.

Exceptions:

qexceptions.ConversionException *(p. **108**)* If this unit is not comparable to **units.ONE** (p. 23).

4.61.2.33 def __rsub__ (*self*, *other*)

Get the difference of another value and this instance of **Quantity** (p. 184).

Parameters:

self

other Another value (not an instance of **Quantity** (p. 184)).

Returns:

A new instance of **Quantity** (p. 184) representing the difference of both quantities.

Exceptions:

qexceptions.ConversionException *(p. **108**)* If the units are not comparable.

4.61.2.34 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

4.61.2.35 def __str__ (*self*)

Get a string describing this **Quantity** (p. 184). The result will be of the form `value unit` (i.e. "12.0 m").

Parameters:

self

Returns:

A string describing this quantity.

4.61.2.36 def __sub__ (*self*, *other*)

Get the difference of another instance of **Quantity** (p. 184) and this instance.

Parameters:

self

other Another instance of **Quantity** (p. 184).

Returns:

A new instance of **Quantity** (p. 184) representing the difference of both quantities.

Exceptions:

qexceptions.ConversionException *(p. **108**)* If the units are not comparable.

4.61.2.37 def __unitComparsion (*unit1*, *unit2*) `[private]`

Helper method.

Parameters:

unit1 A unit.

unit2 Another unit.

Returns:

True if they are compatible and strict is disabled or True if they are equal and strict is enabled.

4.61.2.38 def arccos (*self*)

This method provides the broadcast interface for numpy.arccos.

Parameters:

self

Returns:

The inverse Cosine of this quantity.

Exceptions:

qexceptions.NotDimensionlessException *(*p. *171)* If the unit assigned is not dimensionless.

4.61.2.39 def arccosh (*self*)

This method provides the broadcast interface for numpy.arccosh.

Parameters:

self

Returns:

The inverse hyperbolic Cosine of this quantity.

Exceptions:

qexceptions.NotDimensionlessException *(*p. *171)* If the unit assigned is not dimensionless.

4.61.2.40 def arcsin (*self*)

This method provides the broadcast interface for numpy.arcsin.

Parameters:

self

Returns:

The inverse Sine of this quantity.

Exceptions:

qexceptions.NotDimensionlessException *(*p. *171)* If the unit assigned is not dimensionless.

4.61.2.41 def arcsinh (*self*)

This method provides the broadcast interface for numpy.arcsinh.

Parameters:

self

Returns:

The inverse hyperbolic Sine of this quantity.

Exceptions:

qexceptions.NotDimensionlessException *(*p. *171)* If the unit assigned is not dimensionless.

4.61.2.42 def arctan (*self*)

This method provides the broadcast interface for numpy.arctan.

Parameters:

self

Returns:

The inverse Tangent of this quantity.

Exceptions:

qexceptions.NotDimensionlessException *(*p. *171)* If the unit assigned is not dimensionless.

4.61.2.43 def arctan2 (*self*, *other*)

This method provides the broadcast interface for numpy.arctan2.

Parameters:

self

other Another instance of **Quantity** (p. 184).

Returns:

The inverse two-argument tangent of the arguments.

Exceptions:

qexceptions.NotDimensionlessException *(*p. *171)* If the unit assigned is not dimensionless.

4.61.2.44 def arctanh (*self*)

This method provides the broadcast interface for numpy.arctanh.

Parameters:

self

Returns:

The inverse hyperbolic Tangent of this quantity.

Exceptions:

qexceptions.NotDimensionlessException (p. ***171***) If the unit assigned is not dimensionless.

4.61.2.45 def ceil (*self*)

This method provides the broadcast interface for numpy.ceil.

Parameters:

self

Returns:

The largest integer greater than or equal to this quantity.

4.61.2.46 def conjugate (*self*)

This method provides the broadcast interface for numpy.conjugate.

Parameters:

self

Returns:

This quantity.

4.61.2.47 def cos (*self*)

This method provides the broadcast interface for numpy.cos.

Parameters:

self

Returns:

The Cosine of this quantity.

Exceptions:

qexceptions.NotDimensionlessException (p. ***171***) If the unit assigned is not dimensionless.

4.61.2.48 def cosh (*self*)

This method provides the broadcast interface for numpy.cosh.

Parameters:

self

Returns:

The hyperbolic Cosine of this quantity.

Exceptions:

qexceptions.NotDimensionlessException (p. ***171***) If the unit assigned is not dimensionless.

4.61.2.49 def exp (*self*)

This method provides the broadcast interface for numpy.exp.

Parameters:

self

Returns:

The Exponential of this quantity.

Exceptions:

qexceptions.NotDimensionlessException (p. ***171***) If the unit assigned is not dimensionless.

4.61.2.50 def fabs (*self*)

This method provides the broadcast interface for numpy.fabs.

Parameters:

self

Returns:

The absolute value of this quantity.

4.61.2.51 def floor (*self*)

This method provides the broadcast interface for numpy.floor.

Parameters:

self

Returns:

The largest integer less than or equal to this quantity.

4.61.2.52 def get_default_unit (*self*)

Get the unit that is used commonly for this quantity.

Parameters:

self The current instance of this class.

Returns:

The corresponding unit.

4.61.2.53 def get_value (*self*, *unit*)

Get the absolute value of the quantity using the specified unit.

Parameters:

self The current instance of this class.
unit The unit in which the quantity should be expressed in.

Returns:

The absolute value of the quantity.

Exceptions:

qexceptions.ConversionException (p. ***108***) If the units are not comparable.

4.61.2.54 def hypot (*self*, *other*)

This method provides the broadcast interface for numpy.arctan2.

Parameters:

self
other Another instance of **Quantity** (p. 184).

Returns:

The hypothenusis of the arguments.

4.61.2.55 def is_dimensionless (*self*)

Check if this quantity is dimensionless.

Parameters:

self

Returns:

True, if the unit assigned is comparable to **units.ONE** (p. 23).

4.61.2.56 def is_strict ()

Get the type of quantities calculation. If either strict or non strict evaluation of quantities is implemented.

Returns:

True (i.e. strict enabled) or False (i.e. strict disabled).

4.61.2.57 def log (*self*)

This method provides the broadcast interface for numpy.log.

Parameters:

self

Returns:

The Natural Logarithm of this quantity.

Exceptions:

qexceptions.NotDimensionlessException (p. ***171***) If the unit assigned is not dimensionless.

4.61.2.58 def log10 (*self*)

This method provides the broadcast interface for numpy.log10.

Parameters:

self

Returns:

The decadic Logarithm of this quantity.

Exceptions:

qexceptions.NotDimensionlessException (p. ***171***) If the unit assigned is not dimensionless.

4.61.2.59 def log2 (*self*)

This method provides the broadcast interface for numpy.log2.

Parameters:

self

Returns:

The binary logarithm of this quantity.

Exceptions:

qexceptions.NotDimensionlessException (p. ***171***) If the unit assigned is not dimensionless.

4.61.2.60 def set_strict (*bValue* = `True`)

Turn on/off the strict evaluation of quantities. This will affect any quantities calculation beyond this point.

Parameters:

bValue True or False

4.61.2.61 def sin (*self*)

This method provides the broadcast interface for numpy.sin.

Parameters:

self

Returns:

The Sine of this quantity.

Exceptions:

qexceptions.NotDimensionlessException (p. ***171***) If the unit assigned is not dimensionless.

4.61.2.62 def sinh (*self*)

This method provides the broadcast interface for numpy.sinh.

Parameters:

self

Returns:

The hyperbolic Sine of this quantity.

Exceptions:

qexceptions.NotDimensionlessException (p. ***171***) If the unit assigned is not dimensionless.

4.61.2.63 def sqrt (*self*)

This method provides the broadcast interface for numpy.sqrt.

Parameters:

self

Returns:

The Square Root of this quantity.

4.61.2.64 def square (*self*)

This method provides the broadcast interface for numpy.sqrt.

Parameters:

self

Returns:

The Square Root of this quantity.

4.61.2.65 def tan (*self*)

This method provides the broadcast interface for numpy.tan.

Parameters:

self

Returns:

The Tangent of this quantity.

Exceptions:

qexceptions.NotDimensionlessException (p. ***171***) If the unit assigned is not dimensionless.

4.61.2.66 def tanh (*self*)

This method provides the broadcast interface for numpy.tanh.

Parameters:

self

Returns:

The hyperbolic Tangent of this quantity.

Exceptions:

qexceptions.NotDimensionlessException *(*p. *171)* If the unit assigned is not dimensionless.

4.61.2.67 def value_of (*other*)

Factory for generating quantities.

Parameters:

other A quantity, or another value.

Returns:

A **Quantity** (p. 184). If the argument is a quantity this method returns it. If the argument is a numeric value, this method generates a dimensionless quantity having the argument as value.

Note:

You may use numeric values, sequence types, and instances of **ucomponents.UncertainInput** (p. 309) as values.

4.61.3 Member Data Documentation

4.61.3.1 tuple __accuracy = staticmethod(__accuracy) [static, private]

4.61.3.2 __STRICT = True [static, private]

4.61.3.3 __unit__ [private]

4.61.3.4 tuple __unitComparsion = staticmethod(__unitComparsion) [static, private]

4.61.3.5 __value__ [private]

4.61.3.6 tuple is_strict = staticmethod(is_strict) [static]

4.61.3.7 tuple set_strict = staticmethod(set_strict) [static]

4.61.3.8 tuple value_of = staticmethod(value_of) [static]

4.62 RationalNumber Class Reference

4.62.1 Detailed Description

This class provides support for rational numbers.

Attention:

This class emulates the behaviour of rational numbers. If the overloaded emulation methods have an unknown number type, they fall back to floating point operations.

Note:

Instances of this class can be serialized using pickle.

See also:

RationalNumber.__float__ (p. 216)

Public Member Functions

- def **__abs__**
  This method returns the absolute value of this instance.
- def **__add__**
  Add a number and return the result.
- def **__cmp__**
  Compares this instance to another number.
- def **__coerce__**
  Implementation of coercion rules.
- def **__complex__**
  Cast this rational number to a complex number.
- def **__div__**
  Divide by another number and return the result.
- def **__eq__**

Checks if this instance is equal to a number.

- def **__float__**
  Cast this rational number to a floating point number.
- def **__ge__**
  Checks if this instance is greater or equal to another number.
- def **__getstate__**
  Serialization using pickle.
- def **__gt__**
  Checks if this instance is greater than another number.
- def **__init__**
  Default constructor.
- def **__int__**
  Cast this rational number to an integer.
  Exceptions:
  OverflowError *If the conversion raises an integer overflow.*
- def **__invert__**
  This method returns a new rational number that swapped dividend and divsor of this instance.
- def **__le__**
  Checks if this instance is less or equal to another number.
- def **__long__**
  Cast this rational number to a long integer.
- def **__lt__**
  Checks if this instance is less than another number.
- def **__mul__**
  Multiply a number and return the result.
- def **__ne__**
  Checks if this instance unequal to another number.
- def **__neg__**
  This method returns the negative of this instance.
- def **__nonzero__**
  Check if this instance is nonzero.

- def **__pos__**
  This method returns a copy of this instance.
- def **__pow__**
  Raise this rational number to the given power and return the result.
- def **__radd__**
  Left addition of a numeric value.
- def **__rdiv__**
  Right division of a numeric value.
- def **__rmul__**
  Right multiplication of a numeric value.
- def **__rpow__**
  Raise another value to the power of this rational number. the result.
- def **__rsub__**
  Right substraction of a numeric value.
- def **__setstate__**
  Deserialization using pickle.
- def **__str__**
  This method returns a string representing this rational number.
- def **__sub__**
  Substract a number and return the result.
- def **arccos**
  This method provides the broadcast interface for numpy.arccos.
- def **arccosh**
  This method provides the broadcast interface for numpy.arccosh.
- def **arcsin**
  This method provides the broadcast interface for numpy.arcsin.
- def **arcsinh**
  This method provides the broadcast interface for numpy.arcsinh.
- def **arctan**
  This method provides the broadcast interface for numpy.arctan.

- def **arctan2**
  This method provides the broadcast interface for numpy.arctan2.
- def **arctanh**
  This method provides the broadcast interface for numpy.arctanh.
- def **ceil**
  This method provides the broadcast interface for numpy.ceil.
- def **conjugate**
  This method provides the broadcast interface for numpy.conjugate.
- def **cos**
  This method provides the broadcast interface for numpy.cos.
- def **cosh**
  This method provides the broadcast interface for numpy.cosh.
- def **exp**
  This method provides the broadcast interface for numpy.exp.
- def **fabs**
  This method provides the broadcast interface for numpy.fabs.
- def **floor**
  This method provides the broadcast interface for numpy.floor.
- def **fmod**
  This method provides the broadcast interface for numpy.fmod.
- def **get_dividend**
  Returns the dividend of this instance.
- def **get_divisor**
  Returns the divisor of this instance.
- def **hypot**
  This method provides the broadcast interface for numpy.hypot.
- def **is_integer**
  Check wether this instance could be be casted to long accurately.
- def **log**
  This method provides the broadcast interface for numpy.log.
- def **log10**

This method provides the broadcast interface for numpy.log10.

- def **log2**
  This method provides the broadcast interface for numpy.log2.
- def **normalize**
  This method maintains the canonical form of this rational number and avoids negative divisors.
- def **sin**
  This method provides the broadcast interface for numpy.sin.
- def **sinh**
  This method provides the broadcast interface for numpy.sinh.
- def **sqrt**
  This method provides the broadcast interface for numpy.sqrt.
- def **square**
  This method provides the broadcast interface for numpy.square.
- def **tan**
  This method provides the broadcast interface for numpy.tan.
- def **tanh**
  This method provides the broadcast interface for numpy.tanh.
- def **value_of**
  Factory for generating Rationalnumbers.

Static Public Attributes

- tuple **value_of =** staticmethod(**value_of**)

Private Attributes

- **__dividend__**
- **__divisor__**

4.62.2 Member Function Documentation

4.62.2.1 def __abs__ (*self*)

This method returns the absolute value of this instance.

Parameters:

self

Returns:

A new rational number.

4.62.2.2 def __add__ (*self*, *value*)

Add a number and return the result.

Parameters:

self
value The number to add.

Returns:

The sum of this instance and the argument.

4.62.2.3 def __cmp__ (*self*, *value*)

Compares this instance to another number.

Parameters:

self
value The value to compare to.

Returns:

-1: if this instance is less...; +1: if this is greater than the argument; 0 otherwise

4.62.2.4 def __coerce__ (*self*, *other*)

Implementation of coercion rules.

See also:

Coercion - The page describing the coercion rules.

4.62.2.5 def __complex__ (*self*)

Cast this rational number to a complex number.

Parameters:

self

Returns:

A complex number, having a zero imaginary part.

4.62.2.6 def __div__ (*self*, *value*)

Divide by another number and return the result.

Parameters:

self
value A number.

Returns:

The fraction of this instance and the number.

4.62.2.7 def __eq__ (*self*, *value*)

Checks if this instance is equal to a number.

Parameters:

self
value The value to compare to.

Returns:

If this rational number is equal to the argument.

4.62.2.8 def __float__ (*self*)

Cast this rational number to a floating point number.

Parameters:

self

Returns:

An integer.

4.62.2.9 def __ge__ (*self*, *value*)

Checks if this instance is greater or equal to another number.

Parameters:

self
value The value to compare to.

Returns:

True, if this rational number is greater or equal to the argument.

4.62.2.10 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

4.62.2.11 def __gt__ (*self*, *value*)

Checks if this instance is greater than another number.

Parameters:

self

value The value to compare to.

Returns:

True, if this rational number is greater than the argument.

4.62.2.12 def __init__ (*self*, *dividend*, *divisor* = `1L`)

Default constructor.

This initializes the rational number.

Parameters:

self

dividend An integer representing the dividend of this rational number.

divisor An integer representing the divisor of this rational number. If this parameter is obmitted it is initialized to 1.

4.62.2.13 def __int__ (*self*)

Cast this rational number to an integer.

Exceptions:

OverflowError If the conversion raises an integer overflow.

Parameters:

self

Returns:

An integer.

4.62.2.14 def __invert__ (*self*)

This method returns a new rational number that swapped dividend and divsor of this instance.

Parameters:

self

Returns:

A new rational number.

4.62.2.15 def __le__ (*self*, *value*)

Checks if this instance is less or equal to another number.

Parameters:

self

value The value to compare to.

Returns:

True, if this rational number is less or equal to the argument.

4.62.2.16 def __long__ (*self*)

Cast this rational number to a long integer.

Parameters:

self

Returns:

An integer.

4.62.2.17 def __lt__ (*self*, *value*)

Checks if this instance is less than another number.

Parameters:

self

value The value to compare to.

Returns:

True, if this rational number is less than the argument.

4.62.2.18 def __mul__ (*self*, *value*)

Multiply a number and return the result.

Parameters:

self

value The number to multiply.

Returns:

The product of this instance and the argument.

4.62.2.19 def __ne__ (*self*, *value*)

Checks if this instance unequal to another number.

Parameters:

self

value The value to compare to.

Returns:

True, if this rational number unequal to the argument.

4.62.2.20 def __neg__ (*self*)

This method returns the negative of this instance.

Parameters:

self

Returns:

A new rational number.

4.62.2.21 def __nonzero__ (*self*)

Check if this instance is nonzero.

Parameters:

self

Returns:

True, if the dividend is nonzero.

4.62.2.22 def __pos__ (*self*)

This method returns a copy of this instance.

Parameters:

self

Returns:

A new rational number.

4.62.2.23 def __pow__ (*self*, *value*)

Raise this rational number to the given power and return the result.

Attention:

If the argument is a floating point number then a floating point number will be returned. Note that this may result in a loss of accuracy.

Parameters:

self

value A numeric value representing the power.

Returns:

A new rational number representing power of this instance.

4.62.2.24 def __radd__ (*self*, *value*)

Left addition of a numeric value.

Parameters:

self

value A value to left from this instance.

4.62.2.25 def __rdiv__ (*self*, *value*)

Right division of a numeric value.

Parameters:

self

value A value to left from this instance.

4.62.2.26 def __rmul__ (*self*, *value*)

Right multiplication of a numeric value.

Parameters:

self
value A value to left from this instance.

4.62.2.27 def __rpow__ (*self*, *value*)

Raise another value to the power of this rational number. the result.

Parameters:

self The exponent.
value A value to be raised to the power.

Returns:

A new rational number representing power of this instance.

4.62.2.28 def __rsub__ (*self*, *value*)

Right substraction of a numeric value.

Parameters:

self
value A value to left from this instance.

4.62.2.29 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self
state The state of the object.

4.62.2.30 def __str__ (*self*)

This method returns a string representing this rational number.

Parameters:

self

Returns:

A string representing this rational number.

4.62.2.31 def __sub__ (*self*, *value*)

Substract a number and return the result.

Parameters:

self
value The number to substract.

Returns:

The difference of this instance and the argument.

4.62.2.32 def arccos (*self*)

This method provides the broadcast interface for numpy.arccos.

Parameters:

self

Returns:

The inverse Cosine of this number.

Note:

This number will be converted to float.

4.62.2.33 def arccosh (*self*)

This method provides the broadcast interface for numpy.arccosh.

Parameters:

self

Returns:

The inverse hyperbolic Cosine of this number.

Note:

This number will be converted to float.

4.62.2.34 def arcsin (*self*)

This method provides the broadcast interface for numpy.arcsin.

Parameters:

self

Returns:

The inverse Sine of this number.

Note:

This number will be converted to float.

4.62.2.35 def arcsinh (*self*)

This method provides the broadcast interface for numpy.arcsinh.

Parameters:

self

Returns:

The inverse hyperbolic Sine of this number.

Note:

This number will be converted to float.

4.62.2.36 def arctan (*self*)

This method provides the broadcast interface for numpy.arctan.

Parameters:

self

Returns:

The inverse Tangent of this number.

Note:

This number will be converted to float.

4.62.2.37 def arctan2 (*self*, *other*)

This method provides the broadcast interface for numpy.arctan2.

Parameters:

self
other Another rational number.

Returns:

The binary logarithm of this number.

Note:

This number will be converted to float.

4.62.2.38 def arctanh (*self*)

This method provides the broadcast interface for numpy.arctanh.

Parameters:

self

Returns:

The inverse hyperbolic Tangent of this number.

Note:

This number will be converted to float.

4.62.2.39 def ceil (*self*)

This method provides the broadcast interface for numpy.ceil.

Parameters:

self

Returns:

The largest integer greater than or equal to this number.

Note:

This number will be converted to float.

4.62.2.40 def conjugate (*self*)

This method provides the broadcast interface for numpy.conjugate.

Parameters:

self

Returns:

This number.

Note:

This number will be converted to float.

4.62.2.41 def cos (*self*)

This method provides the broadcast interface for numpy.cos.

Parameters:

self

Returns:

The Cosine of this number.

Note:

This number will be converted to float.

4.62.2.42 def cosh (*self*)

This method provides the broadcast interface for numpy.cosh.

Parameters:

self

Returns:

The hyperbolic Cosine of this number.

Note:

This number will be converted to float.

4.62.2.43 def exp (*self*)

This method provides the broadcast interface for numpy.exp.

Parameters:

self

Returns:

The Exponential of this number.

Note:

This number will be converted to float.

4.62.2.44 def fabs (*self*)

This method provides the broadcast interface for numpy.fabs.

Parameters:

self

Returns:

The absolute value of this number.

Note:

This number will be converted to float.

4.62.2.45 def floor (*self*)

This method provides the broadcast interface for numpy.floor.

Parameters:

self

Returns:

The largest integer less than or equal to this number.

Note:

This number will be converted to float.

4.62.2.46 def fmod (*self*, *other*)

This method provides the broadcast interface for numpy.fmod.

Parameters:

self

other Another value.

Returns:

This number modulo other.

Note:

This number will be converted to float.

Attention:

This method only works one-way. If another type is called i.e. fmod(other, rational(1,1)) then it will fail.

4.62.2.47 def get_dividend (*self*)

Returns the dividend of this instance.

Parameters:

self

Returns:

The dividend of this instance.

4.62.2.48 def get_divisor (*self*)

Returns the divisor of this instance.

Parameters:

self

Returns:

The divisor of this instance.

4.62.2.49 def hypot (*self*, *other*)

This method provides the broadcast interface for numpy.hypot.

Parameters:

self

other Another rational number.

Returns:

The binary logarithm of this number.

Note:

This number will be converted to float.

4.62.2.50 def is_integer (*self*)

Check wether this instance could be be casted to long accurately.

Returns:

True, if the divisor is equal to one.

4.62.2.51 def log (*self*)

This method provides the broadcast interface for numpy.log.

Parameters:

self

Returns:

The Natural Logarithm of this number.

Note:

This number will be converted to float.

4.62.2.52 def log10 (*self*)

This method provides the broadcast interface for numpy.log10.

Parameters:

self

Returns:

The decadic Logarithm of this number.

Note:

This number will be converted to float.

4.62.2.53 def log2 (*self*)

This method provides the broadcast interface for numpy.log2.

Parameters:

self

Returns:

The binary logarithm of this number.

Note:

This number will be converted to float.

4.62.2.54 def normalize (*self*)

This method maintains the canonical form of this rational number and avoids negative divisors.

Parameters:

self

4.62.2.55 def sin (*self*)

This method provides the broadcast interface for numpy.sin.

Parameters:

self

Returns:

The Sine of this number.

Note:

This number will be converted to float.

4.62.2.56 def sinh (*self*)

This method provides the broadcast interface for numpy.sinh.

Parameters:

self

Returns:

The hyperbolic Sine of this number.

Note:

This number will be converted to float.

4.62.2.57 def sqrt (*self*)

This method provides the broadcast interface for numpy.sqrt.

Parameters:

self

Returns:

The Square Root of this number.

Note:

This number will be converted to float.

4.62.2.58 def square (*self*)

This method provides the broadcast interface for numpy.square.

Parameters:

self

Returns:

The binary logarithm of this number.

Note:

This number will be converted to float.

4.62.2.59 def tan (*self*)

This method provides the broadcast interface for numpy.tan.

Parameters:

self

Returns:

The Tangent of this number.

Note:

This number will be converted to float.

4.62.2.60 def tanh (*self*)

This method provides the broadcast interface for numpy.tanh.

Parameters:

self

Returns:

The hyperbolic Tangent of this number.

Note:

This number will be converted to float.

4.62.2.61 def value_of (*number*)

Factory for generating Rationalnumbers.

Parameters:

number a numeric value (not float nor complex)

Exceptions:

TypeError If the argument is not int, long, or a **RationalNumber** (p. 210).

4.62.3 Member Data Documentation

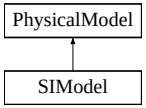
4.62.3.1 __dividend__ [private]

4.62.3.2 __divisor__ [private]

4.62.3.3 tuple value_of = staticmethod(value_of) [static]

4.63 SIModel Class Reference

Inheritance diagram for SIModel::



4.63.1 Detailed Description

The interface for a physical model for SI units.

The basic intend of this class is to provide an mapping between SI base units and physical dimensions.

Public Member Functions

- def **__init__**
  Default constructor.
- def **get_dimension**
  Get the pysical dimension that corresponds to the given SI base unit.

4.63.2 Member Function Documentation

4.63.2.1 def __init__ (*self*)

Default constructor.

Parameters:

self

Reimplemented from **PhysicalModel** (p. 173).

4.63.2.2 def get_dimension (*self*, *unit*)

Get the pysical dimension that corresponds to the given SI base unit.

Parameters:

self

unit The unit to check the dimension for.

Exceptions:

qexceptions.UnknownUnitException *(p. 332)* If the given parameter is no SI base unit.

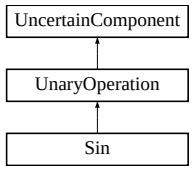
Returns:

The corresponding physical dimension.

Reimplemented from **PhysicalModel** (p. 173).

4.64 Sin Class Reference

Inheritance diagram for Sin::



4.64.1 Detailed Description

This class models the GUM-tree-nodes that take the Sine of a silbling.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = sin(x)$ then the resulting uncertainty is $u(y) = cos(x) \times u(x)$.
- def **get_value**
  Returns the Sine of the silbling.

4.64.2 Member Function Documentation

4.64.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self
right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.64.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self
other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.64.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = sin(x)$ then the resulting uncertainty is $u(y) = cos(x) \times u(x)$.

Parameters:

self
component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.64.2.4 def get_value (*self*)

Returns the Sine of the silbling.

Parameters:

self

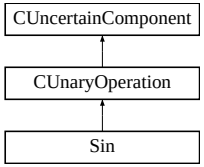
Returns:

A numeric value, representing the Sine of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.65 Sin Class Reference

Inheritance diagram for Sin::



4.65.1 Detailed Description

This class models the sine function.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.65.2 Member Function Documentation

4.65.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.65.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

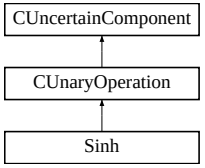
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.66 Sinh Class Reference

Inheritance diagram for Sinh::



4.66.1 Detailed Description

This class models the hyperbolic sine function.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.66.2 Member Function Documentation

4.66.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.66.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

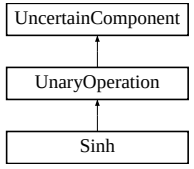
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.67 Sinh Class Reference

Inheritance diagram for Sinh::



4.67.1 Detailed Description

This class models the GUM-tree-nodes that take the Hyperbolic Sine of a silbling.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = sinh(x)$ then the resulting uncertainty is $u(y) = cosh(x)u(x)$.
- def **get_value**
  Returns the Hyperbolic Sine of the silbling.

4.67.2 Member Function Documentation

4.67.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self

right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.67.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.67.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = sinh(x)$ then the resulting uncertainty is $u(y) = cosh(x)u(x)$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.67.2.4 def get_value (*self*)

Returns the Hyperbolic Sine of the silbling.

Parameters:

self

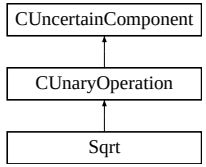
Returns:

A numeric value, representing the Hyperbolic Sine of the silbling.

Reimplemented from **UncertainComponent** (p. 305).

4.68 Sqrt Class Reference

Inheritance diagram for Sqrt::



4.68.1 Detailed Description

This class models taking the square root of an uncertain component.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.68.2 Member Function Documentation

4.68.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self
x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.68.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

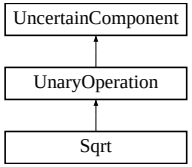
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.69 Sqrt Class Reference

Inheritance diagram for Sqrt::



4.69.1 Detailed Description

This class models the GUM-tree-nodes that take the square root of a silbling.

Public Member Functions

- def **__init__**
  Default constructor.
- def **arithmetic_check**
  Checks for undefined arguments.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = \sqrt{x}$ *then the resulting uncertainty is* $u(y) = \frac{1}{2\sqrt{x}}u(x)$.
- def **get_value**
  Returns the square root of the silbling.

4.69.2 Member Function Documentation

4.69.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self
right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.69.2.2 def arithmetic_check (*self*)

Checks for undefined arguments.

Note:

The square root is only defined for positive values.

Parameters:

self

Exceptions:

ArithmeticError If $x \leq 0$.

Reimplemented from **UncertainComponent** (p. 302).

4.69.2.3 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.69.2.4 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = \sqrt{x}$ then the resulting uncertainty is $u(y) = \frac{1}{2\sqrt{x}}u(x)$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Exceptions:

ZeroDivisionError If the square root is zero.

Reimplemented from **UncertainComponent** (p. 304).

4.69.2.5 def get_value (*self*)

Returns the square root of the silbling.

Parameters:

self

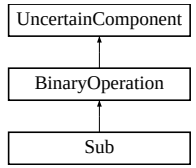
Returns:

A numeric value, representing the square root of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.70 Sub Class Reference

Inheritance diagram for Sub::



4.70.1 Detailed Description

This class models GUM-tree nodes that take the difference of the two silblings.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = x_1 - x_2$ then the resulting uncertainty is $u(y) = u(x_1) - u(x_2)$.
- def **get_value**
  Returns the difference of the silblings assigned.

4.70.2 Member Function Documentation

4.70.2.1 def __init__ (*self*, *left*, *right*)

Default constructor.

Parameters:

self

left Left silbling of this instance.

right Right silbling of this instance.

Reimplemented from **BinaryOperation** (p. 89).

4.70.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **BinaryOperation** (p. 89).

4.70.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = x_1 - x_2$ then the resulting uncertainty is $u(y) = u(x_1) - u(x_2)$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.70.2.4 def get_value (*self*)

Returns the difference of the silblings assigned.

Parameters:

self

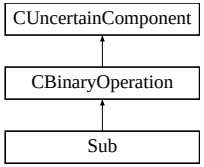
Returns:

A numeric value, representing the difference of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.71 Sub Class Reference

Inheritance diagram for Sub::



4.71.1 Detailed Description

This class models taking the difference of two complex values.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.71.2 Member Function Documentation

4.71.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.71.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

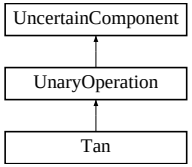
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.72 Tan Class Reference

Inheritance diagram for Tan::



4.72.1 Detailed Description

This class models the GUM-tree-nodes that take the Tangent of a silbling.

Attention:

Because of floating point rounding issues, instances of this class may return invalid values instead of raising an OverflowError for values close to $n \times \frac{\pi}{2}$.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = tan(x)$ then the resulting uncertainty is $u(y) = \frac{u(x)}{cos^2(x)}$.
- def **get_value**
  Returns the Tangent of the silbling.

4.72.2 Member Function Documentation

4.72.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self
right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.72.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self
other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.72.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = tan(x)$ then the resulting uncertainty is $u(y) = \frac{u(x)}{cos^2(x)}$.

Parameters:

self
component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.72.2.4 def get_value (*self*)

Returns the Tangent of the silbling.

Parameters:

self

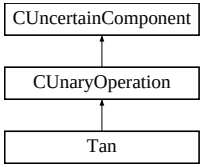
Returns:

A numeric value, representing the Tangent of the silblings.

Reimplemented from **UncertainComponent** (p. 305).

4.73 Tan Class Reference

Inheritance diagram for Tan::



4.73.1 Detailed Description

This class models the tangent function.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.73.2 Member Function Documentation

4.73.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.73.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

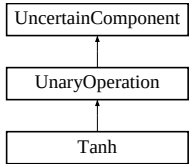
Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.74 Tanh Class Reference

Inheritance diagram for Tanh::



4.74.1 Detailed Description

This class models the GUM-tree-nodes that take the Hyperbolic Tangent of a silbling.

Public Member Functions

- def **__init__**
  Default constructor.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_uncertainty**
  Returns the uncertainty of this node. Let the node represent the operation $y = tanh(x)$ *then the resulting uncertainty is* $u(y) = (1 - tanh^2(x))u(x)$.
- def **get_value**
  Returns the Hyperbolic Tangent of the silbling.

4.74.2 Member Function Documentation

4.74.2.1 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self

right Right silbling of this instance.

Reimplemented from **UnaryOperation** (p. 287).

4.74.2.2 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UnaryOperation** (p. 288).

4.74.2.3 def get_uncertainty (*self*, *component*)

Returns the uncertainty of this node. Let the node represent the operation $y = tanh(x)$ then the resulting uncertainty is $u(y) = (1 - tanh^2(x))u(x)$.

Parameters:

self

component Another instance of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

Reimplemented from **UncertainComponent** (p. 304).

4.74.2.4 def get_value (*self*)

Returns the Hyperbolic Tangent of the silbling.

Parameters:

self

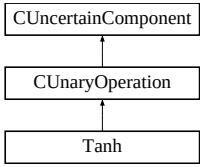
Returns:

A numeric value, representing the Hyperbolic Sine of the silbling.

Reimplemented from **UncertainComponent** (p. 305).

4.75 Tanh Class Reference

Inheritance diagram for Tanh::



4.75.1 Detailed Description

This class models the hyperbolic tangent function.

Public Member Functions

- def **get_uncertainty**
  Get the partial derivate of this component with respect to the given argument.
- def **get_value**
  Get the value of this component.

4.75.2 Member Function Documentation

4.75.2.1 def get_uncertainty (*self*, *x*)

Get the partial derivate of this component with respect to the given argument.

Parameters:

self

x The argument of the partial derivation.

Returns:

The partial derivate.

Reimplemented from **CUncertainComponent** (p. 129).

4.75.2.2 def get_value (*self*)

Get the value of this component.

Parameters:

self

Returns:

The value of this component.

Reimplemented from **CUncertainComponent** (p. 129).

4.76 TestArithmetic Class Reference

4.76.1 Detailed Description

This class provides the tests to verify the rational number module.

Public Member Functions

- def **test_abs**
  Test getting the absolute value of rational numbers.
- def **test_add**
  Test adding instances of the Type ***arithmetic.RationalNumber*** *(*p. *210).*
- def **test_casting**
  Test the casting the Type ***arithmetic.RationalNumber*** *(*p. *210).*
- def **test_comparisions**
  Test the comparision functions of rational numbers.
- def **test_complex_to_matrix**
  Test the conversion from complex numbers to a matrix.
- def **test_div**
  Test dividing instances of the Type ***arithmetic.RationalNumber*** *(*p. *210).*
- def **test_invert**
  Test inverting instances of the Type ***arithmetic.RationalNumber*** *(*p. *210).*
- def **test_mul**
  Test multiplying instances of the Type ***arithmetic.RationalNumber*** *(*p. *210).*
- def **test_neg**
  Test negating instances of the Type ***arithmetic.RationalNumber*** *(*p. *210).*
- def **test_Numpy**
  Test the integration of the Type arithmetic.RationalNumbers in NumPy.

- def **test_pos**
  Test cloning instances of the Type ***arithmetic.RationalNumber*** *(*p. *210).*
- def **test_pow**
  Test powers of instances of the Type ***arithmetic.RationalNumber*** *(*p. *210).*
- def **test_rational_creation**
  Test the creation of the Type ***arithmetic.RationalNumber*** *(*p. *210).*
- def **test_right_ops**
  Test right-operations of the Type ***arithmetic.RationalNumber*** *(*p. *210). Test the operations where an unknown numeric type is a left argument of the instance of rational number.*
- def **test_sub**
  Test substracting instances of the Type ***arithmetic.RationalNumber*** *(*p. *210).*
- def **test_value_of**
  Test the value_of proxy of the Type ***arithmetic.RationalNumber*** *(*p. *210).*

4.76.2 Member Function Documentation

4.76.2.1 def test_abs (*self*)

Test getting the absolute value of rational numbers.

Parameters:

self

4.76.2.2 def test_add (*self*)

Test adding instances of the Type **arithmetic.RationalNumber** (p. 210).

Parameters:

self

4.76.2.3 def test_casting (*self*)

Test the casting the Type **arithmetic.RationalNumber** (p. 210).

Parameters:

self

4.76.2.4 def test_comparisions (*self*)

Test the comparision functions of rational numbers.

Parameters:

self

4.76.2.5 def test_complex_to_matrix (*self*)

Test the conversion from complex numbers to a matrix.

4.76.2.6 def test_div (*self*)

Test dividing instances of the Type **arithmetic.RationalNumber** (p. 210).

Parameters:

self

4.76.2.7 def test_invert (*self*)

Test inverting instances of the Type **arithmetic.RationalNumber** (p. 210).

Parameters:

self

4.76.2.8 def test_mul (*self*)

Test multiplying instances of the Type **arithmetic.RationalNumber** (p. 210).

Parameters:

self

4.76.2.9 def test_neg (*self*)

Test negating instances of the Type **arithmetic.RationalNumber** (p. 210).

Parameters:

self

4.76.2.10 def test_Numpy (*self*)

Test the integration of the Type arithmetic.RationalNumbers in NumPy.

4.76.2.11 def test_pos (*self*)

Test cloning instances of the Type **arithmetic.RationalNumber** (p. 210).

Parameters:

self

4.76.2.12 def test_pow (*self*)

Test powers of instances of the Type **arithmetic.RationalNumber** (p. 210).

Parameters:

self

4.76.2.13 def test_rational_creation (*self*)

Test the creation of the Type **arithmetic.RationalNumber** (p. 210).

Parameters:

self

4.76.2.14 def test_right_ops (*self*)

Test right-operations of the Type **arithmetic.RationalNumber** (p. 210). Test the operations where an unknown numeric type is a left argument of the instance of rational number.

Attention:

This test is not as strict as the individual tests for the operations that require a left argument. This is because the functions tested here rely on them.

Parameters:

self

4.76.2.15 def test_sub (*self*)

Test substracting instances of the Type **arithmetic.RationalNumber** (p. 210).

Parameters:

self

4.76.2.16 def test_value_of (*self*)

Test the value_of proxy of the Type **arithmetic.RationalNumber** (p. 210).

Parameters:

self

4.77 TestComplexUncertaintyComponents Class Reference

4.77.1 Detailed Description

This class provides test-cases for the Module cucomponents.

See also:

cucomponents (p. 26)

Public Member Functions

- def **setUp**
  This method sets up the testcase for every individual test.

Private Attributes

- **__input_1**
- **__input_2**

Classes

- class **OperationTest**
  This is the abstract super class for testing all operations of the Module cucomponents.

4.77.2 Member Function Documentation

4.77.2.1 def setUp (*self*)

This method sets up the testcase for every individual test.

Parameters:

self

4.77.3 Member Data Documentation

4.77.3.1 __input_1 [private]

4.77.3.2 __input_2 [private]

4.78 TestComplexUncertaintyComponents::OperationTest Class Reference

4.78.1 Detailed Description

This is the abstract super class for testing all operations of the Module cucomponents.

See also:

cucomponents (p. 26)

Public Member Functions

- def **__init__**
  The Default Constructor.
- def **get_component**
  This method returns the component to be tested.
- def **get_max_error**
  This method returns the maximum allowable error.
- def **test_dependencies**
  This method checks wheter the dependencies are carried out correctly.
- def **test_type**
  This method checks for the type.
- def **test_uncertainty**
  This method checks for the correct value.
- def **test_value**
  This method checks for the correct uncertainty propagation.

Private Attributes

- **__component**
- **__dependents**
- **__max_err**
- **__type**
- **__uncertainty**
- **__value**

4.78.2 Member Function Documentation

4.78.2.1 def __init__ (*self*, *component*, *type*, *value*, *uncertainty*, *dependents*, *max_err* = `1e-6`)

The Default Constructor.

Parameters:

self

component The component to test.

type The type of the component.

value The expected value of the component.

uncertainty The expected uncertainty of the component.

dependents A list of components this component depends on.

max_err The maximum acceptable numeric error.

4.78.2.2 def get_component (*self*)

This method returns the component to be tested.

Parameters:

self

Returns:

The component that is currently tested.

4.78.2.3 def get_max_error (*self*)

This method returns the maximum allowable error.

Parameters:

self

Returns:

The maximum allowable error.

4.78.2.4 def test_dependencies (*self*)

This method checks wheter the dependencies are carried out correctly.

Parameters:

self

4.78.2.5 def test_type (*self*)

This method checks for the type.

Parameters:

self

4.78.2.6 def test_uncertainty (*self*)

This method checks for the correct value.

Parameters:

self

4.78.2.7 def test_value (*self*)

This method checks for the correct uncertainty propagation.

Parameters:

self

4.78.3 Member Data Documentation

4.78.3.1 __component `[private]`

4.78.3.2 __dependents `[private]`

4.78.3.3 __max_err `[private]`

4.78.3.4 __type `[private]`

4.78.3.5 __uncertainty `[private]`

4.78.3.6 __value `[private]`

4.79 TestGUMTree Class Reference

4.79.1 Detailed Description

These classes test the function of the global elements of the GUM-tree, namely the Context class.

Public Member Functions

- def **test_automatic_differentiation_example**
  Check the Module ucomponents by evaluating an example from a paper.
- def **test_correlations**
  Test correlating scalar uncertain components.
- def **test_GUM_example**
  Check the Module ucomponents by evaluating another GUM-example.
- def **test_GUM_integration**
  Check the Module ucomponents by evaluating a GUM-example.
- def **test_GUM_tree_example**
  Check the Module ucomponents by evaluating another GUM-example.
- def **testByGUMComplexExample**
  Check the Module ucomponents by evaluating a ByGUM-example.
- def **testByGUMComplexExampleU**
  Check the Module ucomponents by evaluating a ByGUM-example using units.

4.79.2 Member Function Documentation

4.79.2.1 def test_automatic_differentiation_example (*self*)

Check the Module ucomponents by evaluating an example from a paper.

Parameters:

self

See also:

"Calculating measurement uncertainty using automatic differentiation"; B.D.Hall; Measurement Science Technology Issue 13 (2002)

4.79.2.2 def test_correlations (*self*)

Test correlating scalar uncertain components.

Parameters:

self

See also:

ucomponents.Context.set_correlation (p. 103)
ucomponents.Context.get_correlation (p. 103)

4.79.2.3 def test_GUM_example (*self*)

Check the Module ucomponents by evaluating another GUM-example.

Parameters:

self

See also:

ISO GUM

4.79.2.4 def test_GUM_integration (*self*)

Check the Module ucomponents by evaluating a GUM-example.

Parameters:

self

See also:

"Guidlines for Evaluating and Expressing the uncertainty in Measurements"; B.N.Taylor and C.E. Kuyatt; NIST 1297 (1994)

4.79.2.5 def test_GUM_tree_example (*self*)

Check the Module ucomponents by evaluating another GUM-example.

Parameters:

self

See also:

"Guidlines for Evaluating and Expressing the uncertainty in Measurements"; B.N.Taylor and C.E. Kuyatt; NIST 1297 (1994)

4.79.2.6 def testByGUMComplexExample (*self*)

Check the Module ucomponents by evaluating a ByGUM-example.

Parameters:

self

See also:

"ByGUM: A Python software package for calculating measurement uncertainty"; B. D. Hall; Industral Research Limited Report 1305; 2005

4.79.2.7 def testByGUMComplexExampleU (*self*)

Check the Module ucomponents by evaluating a ByGUM-example using units.

Parameters:

self

See also:

"ByGUM: A Python software package for calculating measurement uncertainty"; B. D. Hall; Industral Research Limited Report 1305; 2005

4.80 TestOperators Class Reference

4.80.1 Detailed Description

Test the unit conversion operators.

Public Member Functions

- def **test_add_operator**
  Test the unit add operator.
- def **test_compound_operations**
  Test all permuations of binary operations on unit operators.
- def **TEST_CONV**
  Test an Operator with an integer/long input and an expected output value. Also make sure, that the output-type is correct.
- def **TEST_CONV_APPX**
  Test an Operator with an integer/long input and an expected output value. Also make sure, that the output-type is correct.
- def **test_identity**
  Test the global identity variable (for unit converters).
- def **test_log_exp_operator**
  Test the exponential and logarithmic unit operators.
- def **test_multiply_operator**
  Test the unit multiply operator.

Static Public Attributes

- tuple **TEST_CONV** = staticmethod(**TEST_CONV**)
- tuple **TEST_CONV_APPX** = staticmethod(**TEST_CONV_APPX**)

4.80.2 Member Function Documentation

4.80.2.1 def test_add_operator (*self*)

Test the unit add operator.

Parameters:

self

4.80.2.2 def test_compound_operations (*self*)

Test all permuations of binary operations on unit operators.

Parameters:

self

4.80.2.3 def TEST_CONV (*operator*, *invalue*, *outvalue*, *outtype*)

Test an Operator with an integer/long input and an expected output value. Also make sure, that the output-type is correct.

4.80.2.4 def TEST_CONV_APPX (*operator*, *invalue*, *outvalue*, *outtype*, *confidence*)

Test an Operator with an integer/long input and an expected output value. Also make sure, that the output-type is correct.

4.80.2.5 def test_identity (*self*)

Test the global identity variable (for unit converters).

Parameters:

self

4.80.2.6 def test_log_exp_operator (*self*)

Test the exponential and logarithmic unit operators.

Parameters:

self

4.80.2.7 def test_multiply_operator (*self*)

Test the unit multiply operator.

Parameters:

self

4.80.3 Member Data Documentation

4.80.3.1 tuple TEST_CONV = staticmethod(TEST_CONV) [static]

4.80.3.2 tuple TEST_CONV_APPX = staticmethod(TEST_CONV_APPX) [static]

4.81 TestQuantity Class Reference

4.81.1 Detailed Description

This class provides the test cases for the quantities.

Public Member Functions

- def **setUp**
  This method initializes this test instance.
- def **test_abs**
  Test getting absolute values of quantities.
- def **test_absolute**
  Test the operator numpy.absolute on quantities.
- def **test_add**
  Test adding quantities.
- def **test_arccos**
  Test the operator numpy.arccos on quantities.
- def **test_arccosh**
  Test the operator numpy.arccosh on quantities.
- def **test_arcsin**
  Test the operator numpy.arcsin on quantities.
- def **test_arcsinh**
  Test the operator numpy.arcsinh on quantities.

- def **test_arctan**
  Test the operator numpy.arctan on quantities.
- def **test_arctan2**
  Test the operator numpy.arctan2 on quantities.
- def **test_arctanh**
  Test the operator numpy.arctanh on quantities.
- def **test_casts**
  Test casting quatities to other numeric types.
- def **test_ceil**
  Test the operator numpy.ceil on quantities.
- def **test_comparisions**
  Test comparing quantities.
- def **test_conjugate**
  Test the operator numpy.conjugate on quantities.
- def **test_cos**
  Test the operator numpy.cos on quantities.
- def **test_cosh**
  Test the operator numpy.cosh on quantities.
- def **test_div**
  Test dividing quantities.
- def **test_exp**
  Test the operator numpy.exp on quantities.
- def **test_fabs**
  Test the operator numpy.fabs on quantities.
- def **test_floor**
  Test the operator numpy.floor on quantities.
- def **test_fmod**
  Test the operator numpy.fmod on quantities.
- def **test_hypot**
  Test the operator numpy.hypot on quantities.

- def **test_iadd**
  Test augmented-add (+=) of quantities.
- def **test_idiv**
  Test augmented-division (/=) of quantities.
- def **test_imul**
  Test augmented-multiply (∗=) of quantities.
- def **test_init**
  Test the initialization of quantities.
- def **test_invert**
  Test inverting of quantities.
- def **test_ipow**
  Test augmented-powers (∗∗=) of quantities.
- def **test_isub**
  Test augmented-subtract (-=) of quantities.
- def **test_log**
  Test the operator numpy.log on quantities.
- def **test_log10**
  Test the operator numpy.log10 on quantities.
- def **test_log2**
  Test the operator numpy.log2 on quantities.
- def **test_mul**
  Test multiplying quantities.
- def **test_neg**
  Test negating quantities.
- def **test_pos**
  Test cloning quantities.
- def **test_pow**
  Test powers of quantities.
- def **test_radd**
  Test right-add of quantities.
- def **test_rdiv**

Test right-divide of quantities.

- def **test_rmul**
  Test right-multiply of quantities.
- def **test_rpow**
  Test right-powers of quantities.
- def **test_rsub**
  Test right-subtract of quantities.
- def **test_sin**
  Test the operator numpy.sin on quantities.
- def **test_sinh**
  Test the operator numpy.sinh on quantities.
- def **test_sqrt**
  Test the operator numpy.sqrt on quantities.
- def **test_square**
  Test the operator numpy.square on quantities.
- def **test_sub**
  Test subtracting quantities.
- def **test_tan**
  Test the operator numpy.tan on quantities.
- def **test_tanh**
  Test the operator numpy.tanh on quantities.

Public Attributes

- **dimensionless**
- **incompat**
- **newtons1**
- **newtons2**
- **other**
- **otherStr**

4.81.2 Member Function Documentation

4.81.2.1 def setUp (*self*)

This method initializes this test instance.

Parameters:

self

4.81.2.2 def test_abs (*self*)

Test getting absolute values of quantities.

Parameters:

self

4.81.2.3 def test_absolute (*self*)

Test the operator numpy.absolute on quantities.

Parameters:

self

4.81.2.4 def test_add (*self*)

Test adding quantities.

Parameters:

self

4.81.2.5 def test_arccos (*self*)

Test the operator numpy.arccos on quantities.

Parameters:

self

4.81.2.6 def test_arccosh (*self*)

Test the operator numpy.arccosh on quantities.

Parameters:

self

4.81.2.7 def test_arcsin (*self*)

Test the operator numpy.arcsin on quantities.

Parameters:

self

4.81.2.8 def test_arcsinh (*self*)

Test the operator numpy.arcsinh on quantities.

Parameters:

self

4.81.2.9 def test_arctan (*self*)

Test the operator numpy.arctan on quantities.

Parameters:

self

4.81.2.10 def test_arctan2 (*self*)

Test the operator numpy.arctan2 on quantities.

Parameters:

self

4.81.2.11 def test_arctanh (*self*)

Test the operator numpy.arctanh on quantities.

Parameters:

self

4.81.2.12 def test_casts (*self*)

Test casting quatities to other numeric types.

Parameters:

self

4.81.2.13 def test_ceil (*self*)

Test the operator numpy.ceil on quantities.

Parameters:

self

4.81.2.14 def test_comparisions (*self*)

Test comparing quantities.

Parameters:

self

4.81.2.15 def test_conjugate (*self*)

Test the operator numpy.conjugate on quantities.

Parameters:

self

4.81.2.16 def test_cos (*self*)

Test the operator numpy.cos on quantities.

Parameters:

self

4.81.2.17 def test_cosh (*self*)

Test the operator numpy.cosh on quantities.

Parameters:

self

4.81.2.18 def test_div (*self*)

Test dividing quantities.

Parameters:

self

4.81.2.19 def test_exp (*self*)

Test the operator numpy.exp on quantities.

Parameters:

self

4.81.2.20 def test_fabs (*self*)

Test the operator numpy.fabs on quantities.

Parameters:

self

4.81.2.21 def test_floor (*self*)

Test the operator numpy.floor on quantities.

Parameters:

self

4.81.2.22 def test_fmod (*self*)

Test the operator numpy.fmod on quantities.

Parameters:

self

4.81.2.23 def test_hypot (*self*)

Test the operator numpy.hypot on quantities.

Parameters:

self

4.81.2.24 def test_iadd (*self*)

Test augmented-add (+=) of quantities.

Parameters:

self

4.81.2.25 def test_idiv (*self*)

Test augmented-division (/=) of quantities.

Parameters:

self

4.81.2.26 def test_imul (*self*)

Test augmented-multiply (∗=) of quantities.

Parameters:

self

4.81.2.27 def test_init (*self*)

Test the initialization of quantities.

Parameters:

self

4.81.2.28 def test_invert (*self*)

Test inverting of quantities.

Parameters:

self

4.81.2.29 def test_ipow (*self*)

Test augmented-powers (∗∗=) of quantities.

Parameters:

self

4.81.2.30 def test_isub (*self*)

Test augmented-subtract (-=) of quantities.

Parameters:

self

4.81.2.31 def test_log (*self*)

Test the operator numpy.log on quantities.

Parameters:

self

4.81.2.32 def test_log10 (*self*)

Test the operator numpy.log10 on quantities.

Parameters:

self

4.81.2.33 def test_log2 (*self*)

Test the operator numpy.log2 on quantities.

Parameters:

self

4.81.2.34 def test_mul (*self*)

Test multiplying quantities.

Parameters:

self

4.81.2.35 def test_neg (*self*)

Test negating quantities.

Parameters:

self

4.81.2.36 def test_pos (*self*)

Test cloning quantities.

Parameters:

self

4.81.2.37 def test_pow (*self*)

Test powers of quantities.

Parameters:

self

4.81.2.38 def test_radd (*self*)

Test right-add of quantities.

Parameters:

self

4.81.2.39 def test_rdiv (*self*)

Test right-divide of quantities.

Parameters:

self

4.81.2.40 def test_rmul (*self*)

Test right-multiply of quantities.

Parameters:

self

4.81.2.41 def test_rpow (*self*)

Test right-powers of quantities.

Parameters:

self

4.81.2.42 def test_rsub (*self*)

Test right-subtract of quantities.

Parameters:

self

4.81.2.43 def test_sin (*self*)

Test the operator numpy.sin on quantities.

Parameters:

self

4.81.2.44 def test_sinh (*self*)

Test the operator numpy.sinh on quantities.

Parameters:

self

4.81.2.45 def test_sqrt (*self*)

Test the operator numpy.sqrt on quantities.

Parameters:

self

4.81.2.46 def test_square (*self*)

Test the operator numpy.square on quantities.

Parameters:

self

4.81.2.47 def test_sub (*self*)

Test subtracting quantities.

Parameters:

self

4.81.2.48 def test_tan (*self*)

Test the operator numpy.tan on quantities.

Parameters:

self

4.81.2.49 def test_tanh (*self*)

Test the operator numpy.tanh on quantities.

Parameters:

self

4.81.3 Member Data Documentation

4.81.3.1 dimensionless

4.81.3.2 incompat

4.81.3.3 newtons1

4.81.3.4 newtons2

4.81.3.5 other

4.81.3.6 otherStr

4.82 TestSIUnits Class Reference

4.82.1 Detailed Description

SI Testing class. This class tests the definition and semantics of the SI units.

Public Member Functions

- def **ALTERNATE_TEST**
  Test alternate units.
- def **BASE_UNIT_TEST**
  Test base units.
- def **test_alternate_units**
  Test the alternate SI units.
- def **test_base_units**
  Test the base SI units.
- def **test_rational_powers**
  Test rational powers of units.

- def **test_transformed_units**
  Test the transformed SI units (i.e. there is only one: degrees Celsius).
- def **TRANSFORMED_TEST**
  Test transformed units.

Static Public Attributes

- tuple **ALTERNATE_TEST** = staticmethod(**ALTERNATE_TEST**)
- tuple **BASE_UNIT_TEST** = staticmethod(**BASE_UNIT_TEST**)
- tuple **TRANSFORMED_TEST** = staticmethod(**TRANSFORMED_TEST**)

4.82.2 Member Function Documentation

4.82.2.1 def ALTERNATE_TEST (*unit*, *parent*, *idiotsUnit*, *symbol*)

Test alternate units.

Parameters:

unit The instance of the unit to test.

parent The expected parent unit of the unit.

idiotsUnit A unit that should not be compatible or equal to this unit.

symbol The string is equal to the units symbol.

4.82.2.2 def BASE_UNIT_TEST (*unit*, *dimension*, *idiotsUnit*, *symbol*, *idiotsDimension*)

Test base units.

Parameters:

unit The instance of the unit to test.

dimension The physical dimension in which the base unit should be defined.

idiotsUnit A unit that should not be compatible or equal to this unit.

symbol The string is equal to the units symbol.

idiotsDimension A dimension in which the unit should not be defined.

4.82.2.3 def test_alternate_units (*self*)

Test the alternate SI units.

Parameters:

self

See also:

ALTERNATE_TEST (p. 278)

4.82.2.4 def test_base_units (*self*)

Test the base SI units.

Parameters:

self

See also:

BASE_UNIT_TEST (p. 278)

4.82.2.5 def test_rational_powers (*self*)

Test rational powers of units.

Parameters:

self

4.82.2.6 def test_transformed_units (*self*)

Test the transformed SI units (i.e. there is only one: degrees Celsius).

Parameters:

self

See also:

TRANSFORMED_TEST (p. 278)

4.82.2.7 def TRANSFORMED_TEST (*unit*, *parent*, *valueParent*, *valueTransformed*, *maxAcceptableError*)

Test transformed units.

Parameters:

unit The instance of the unit to test.

parent The expected parent unit of the unit.

valueParent An example for a numeric value that represents the parent.

valueTransformed An example for a numeric value that represens this unit.

maxAcceptableError The maximum acceptable numeric error.

4.82.3 Member Data Documentation

4.82.3.1 tuple ALTERNATE_TEST = staticmethod(ALTERNATE_TEST) [static]

4.82.3.2 tuple BASE_UNIT_TEST = staticmethod(BASE_UNIT_TEST) [static]

4.82.3.3 tuple TRANSFORMED_TEST = staticmethod(TRANSFORMED_TEST) [static]

4.83 TestUncertaintyComponents Class Reference

4.83.1 Detailed Description

This class provides tests for the ucomponents module.

Public Member Functions

- def **test_clear_duplicates**
  Test the function ***ucomponents.clearDuplicates*** *(p. 20).*
- def **TEST_COMPONENT_SERIALIZATION**
  Check serializing components of uncertainty.
- def **TEST_UNCERTAIN_COMPONENT**
  A general component test for ***ucomponents.UncertainComponent*** *(p. 289).*

Public Attributes

- **inputFloat**
- **inputLong**
- **inputRational**

Static Public Attributes

- tuple **TEST_COMPONENT_SERIALIZATION** = staticmethod(**TEST_COMPONENT_SERIALIZATION**)
- tuple **TEST_UNCERTAIN_COMPONENT** = staticmethod(**TEST_UNCERTAIN_COMPONENT**)

Classes

- class **Element**

  A class that is used for testing for identity.

4.83.2 Member Function Documentation

4.83.2.1 def test_clear_duplicates (*self*)

Test the function **ucomponents.clearDuplicates** (p. 20).

Parameters:

self

4.83.2.2 def TEST_COMPONENT_SERIALIZATION (*component*, *type*, *numsilblings*, *protocol* = `pickle.HIGHEST_PROTOCOL`)

Check serializing components of uncertainty.

Parameters:

component The instance of **ucomponents.UncertainComponent** (p. 289) to check.

type The type of the component.

numsilblings The number components that is returned by **ucomponents.UncertainComponent.depends_on** (p. 303)

protocol The serialization protocol version to use.

See also:

pickle.HIGHEST_PROTOCOL

Returns:

The deserialized instance.

4.83.2.3 def TEST_UNCERTAIN_COMPONENT (*component*, *type*, *expectedValue*, *expectedUncertainty*, *accuracy*)

A general component test for **ucomponents.UncertainComponent** (p. 289).

4.83.3 Member Data Documentation

4.83.3.1 inputFloat

4.83.3.2 inputLong

4.83.3.3 inputRational

4.83.3.4 tuple TEST_COMPONENT_SERIALIZATION = staticmethod(TEST_COMPONENT_SERIALIZATION) `[static]`

4.83.3.5 tuple TEST_UNCERTAIN_COMPONENT = staticmethod(TEST_UNCERTAIN_COMPONENT) `[static]`

4.84 TestUncertaintyComponents::Element Class Reference

4.84.1 Detailed Description

A class that is used for testing for identity.

Public Member Functions

- def **__eq__**

  The function that is normally used for comparision.

- def **__hash__**

  A necessary method for working with containers.

- def **__init__**

  Default constructor.

- def **get_value**

  Return the value assigned.

Public Attributes

- **value**

4.84.2 Member Function Documentation

4.84.2.1 def __eq__ (*self*, *other*)

The function that is normally used for comparision.

Attention:

When using **ucomponents.clearDuplicates** (p. 20) this method should not get called.

Parameters:

self
other Another instance of **Element** (p. 280).

4.84.2.2 def __hash__ (*self*)

A necessary method for working with containers.

Parameters:

self

4.84.2.3 def __init__ (*self*, *value*)

Default constructor.

Parameters:

self
value A value to assign to the instance

4.84.2.4 def get_value (*self*)

Return the value assigned.

Parameters:

self

Returns:

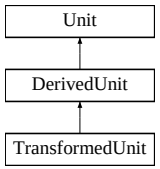
The value assigned

4.84.3 Member Data Documentation

4.84.3.1 value

4.85 TransformedUnit Class Reference

Inheritance diagram for TransformedUnit::



4.85.1 Detailed Description

This class provides an interface for a unit that has been derived from a unit using an operator.

For example feet can be derived from meter.

See also:

Unit.__mul__ (p. 318)
Unit.__div__ (p. 317)
Unit.__add__ (p. 316)
Unit.__sub__ (p. 320)

Note:

Instances of this class can be serialized using pickle.

Public Member Functions

- def **__eq__**

  Compare two transformed units. Two transformed units are equal, if their transformation as well as their parent units are equal.

- def **__getstate__**

  Serialization using pickle.

- def **__init__**

  Default constructor.

- def **__setstate__**

  Deserialization using pickle.

- def **__str__**

  Print the current unit. This function returns a string of the form $(op\,(parentunit)$*, for example* $(K + 273.15)$ *for degrees celsius transformed from kelvins.*

- def **get_parent**

  Return the parent unit.

- def **get_system_unit**

  Get the corresponding system unit.

- def **to_parent_unit**

  Get the operator to convert to the parent unit.

- def **to_system_unit**

  Get the operator to convert to the corresponding system unit.

Static Private Attributes

- **__operator__** = None
  What is the operator to the parent unit.
- **__parentUnit__** = None
  What was the original unit.

4.85.2 Member Function Documentation

4.85.2.1 def __eq__ (*self*, *other*)

Compare two transformed units. Two transformed units are equal, if their transformation as well as their parent units are equal.

Parameters:

self
other Another instance of a transformed unit.

Returns:

True, if the units are equal.

Reimplemented from **Unit** (p. 317).

4.85.2.2 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **Unit** (p. 317).

4.85.2.3 def __init__ (*self*, *parent*, *operator*)

Default constructor.

Parameters:

self
parent The parent unit of the current unit.
operator The operator that forms this unit from the parent unit.

4.85.2.4 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self
state The state of the object.

Reimplemented from **Unit** (p. 320).

4.85.2.5 def __str__ (*self*)

Print the current unit. This function returns a string of the form $(op\,(parentunit)$, for example $(K + 273.15)$ for degrees celsius transformed from kelvins.

Parameters:

self

Returns:

A string describing this unit.

See also:

__ProductElement.__str__
operators.UnitOperator.__str__ (p. 328)

Reimplemented from **Unit** (p. 320).

4.85.2.6 def get_parent (*self*)

Return the parent unit.

Parameters:

self

Returns:

The unit before the transformation.

4.85.2.7 def get_system_unit (*self*)

Get the corresponding system unit.

Parameters:

self

Returns:

The corresponding system unit.

Reimplemented from **Unit** (p. 322).

4.85.2.8 def to_parent_unit (*self*)

Get the operator to convert to the parent unit.

Parameters:

self

Returns:

The operator to the parent unit.

4.85.2.9 def to_system_unit (*self*)

Get the operator to convert to the corresponding system unit.

Parameters:

self

Returns:

The operator to the system unit.

Reimplemented from **Unit** (p. 324).

4.85.3 Member Data Documentation

4.85.3.1 __operator__ = None `[static, private]`

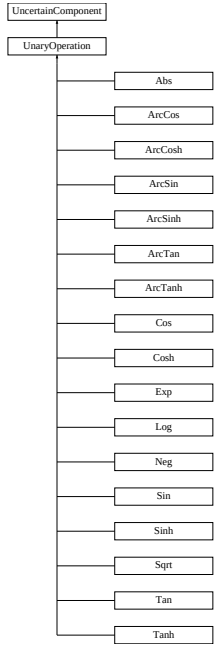
What is the operator to the parent unit.

4.85.3.2 __parentUnit__ = None `[static, private]`

What was the original unit.

4.86 UnaryOperation Class Reference

Inheritance diagram for UnaryOperation::



4.86.1 Detailed Description

The abstract base class for modelling unary operations. This class provides the abstract interface for GUM-tree-nodes that have one silbling.

Public Member Functions

- def **__getstate__**
  Serialization using pickle.
- def **__init__**
  Default constructor.
- def **__setstate__**
  Deserialization using pickle.
- def **depends_on**

Abstract method: The implementation should return a list of the components of uncertainty, that this component depends on.

- def **equal_debug**

  A method that is only used for serialization checking.

- def **get_silbling**

  Return the silbling.

Static Private Attributes

- **__right** = None

  The silbling of the operation.

4.86.2 Member Function Documentation

4.86.2.1 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **UncertainComponent** (p. 295).

4.86.2.2 def __init__ (*self*, *right*)

Default constructor.

Parameters:

self

right The silbling of this instance.

Reimplemented in **Cos** (p. 110), **Sin** (p. 233), **Tan** (p. 245), **Sqrt** (p. 240), **Log** (p. 153), **ArcSin** (p. 68), **ArcSinh** (p. 73), **ArcCos** (p. 62), **ArcCosh** (p. 65), **ArcTan** (p. 75), **ArcTanh** (p. 81), **Cosh** (p. 114), **Sinh** (p. 237), **Tanh** (p. 248), **Exp** (p. 146), **Abs** (p. 47), and **Neg** (p. 169).

4.86.2.3 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

Reimplemented from **UncertainComponent** (p. 299).

4.86.2.4 def depends_on (*self*)

Abstract method: The implementation should return a list of the components of uncertainty, that this component depends on.

Returns:

A list of the components of uncertainty.

Reimplemented from **UncertainComponent** (p. 303).

4.86.2.5 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UncertainComponent** (p. 303).

Reimplemented in **Cos** (p. 110), **Sin** (p. 233), **Tan** (p. 246), **Sqrt** (p. 241), **Log** (p. 154), **ArcSin** (p. 69), **ArcSinh** (p. 73), **ArcCos** (p. 63), **ArcCosh** (p. 65), **ArcTan** (p. 75), **ArcTanh** (p. 81), **Cosh** (p. 114), **Sinh** (p. 237), **Tanh** (p. 249), **Exp** (p. 146), **Abs** (p. 48), and **Neg** (p. 169).

4.86.2.6 def get_silbling (*self*)

Return the silbling.

Returns:

The silbling.

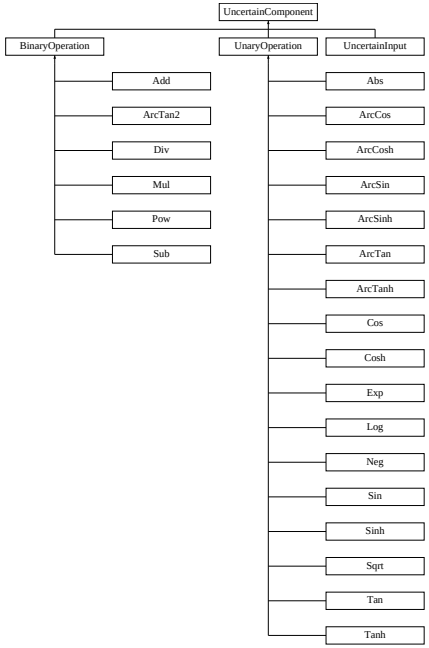
4.86.3 Member Data Documentation

4.86.3.1 __right = None [static, private]

The silbling of the operation.

4.87 UncertainComponent Class Reference

Inheritance diagram for UncertainComponent::



4.87.1 Detailed Description

This is the abstract base class to model components of uncertainty as described in by "The GUM Tree".

See also:

"The "GUM Tree": A software design pattern for handling measurement uncertainty"; B. D. Hall; Industrial Research Report 1291; Measurements Standards

Laboratory New Zealand (2003).

Public Member Functions

- def **__abs__**
  This method returs the absolute value of this instance.
- def **__add__**
  This method adds the argument to this instance.
- def **__coerce__**
  Implementation of coercion rules.
- def **__div__**
  This method divides this instance by the argument.
- def **__eq__**
  This method is an alias for (self is other). It checks if the argument is identical with the current instance.
- def **__getstate__**
  Serialization using pickle.
- def **__init__**
  Default constructor.
- def **__invert__**
  Inverts this instance.
- def **__mul__**
  This method multiplies the argument by this instance.
- def **__ne__**
  This method is an alias for not(self is other). It checks if the argument is not identical with the current instance.
- def **__neg__**
  This method negates this instance.
- def **__pow__**
  This method raises this to the power of the argument.
- def **__radd__**
  This method adds this instance to the argument.
- def **__rdiv__**

This method divides the argument by this instance.

- def **__rmul__**

  This method multiplies the argument by this instance.

- def **__rpow__**

  This method raises the argument to the power of this instance.

- def **__rsub__**

  This method substracts this instance from the argument.

- def **__setstate__**

  Deserialization using pickle.

- def **__str__**

  This method returs the absolute value of this instance.

- def **__sub__**

  This method substracts the argument from this instance.

- def **arccos**

  This method provides the broadcast interface for numpy.arccos.

- def **arccosh**

  This method provides the broadcast interface for numpy.arccosh.

- def **arcsin**

  This method provides the broadcast interface for numpy.arcsin.

- def **arcsine**

  A factory method, that can be used to create instances of uncertain components. This method returns uncertain inputs that are quantified as an arcsin distribution.

- def **arcsinh**

  This method provides the broadcast interface for numpy.arcsinh.

- def **arctan**

  This method provides the broadcast interface for numpy.arctan.

- def **arctan2**

  This method provides an interface for numpy.arctan2.

- def **arctanh**

  This method provides the broadcast interface for numpy.arctanh.

- def **arithmetic_check**

This method checks this instance for mathematical correctness. You should overload this method, if your class is not defined for specific argument values. If any (mathematical) invalid values have been assigned, your implementation should raise an ArithmeticError explaining the problem. This method is usually called within the constructor of a class, after the members have been initialized.

- def **beta**

  A factory method, that can be used to create instances of uncertain components. This method returns uncertain inputs that are quantified as a beta distribution, having the parameters p and q.

- def **cos**

  This method provides the broadcast interface for numpy.cos.

- def **cosh**

  This method provides the broadcast interface for numpy.cosh.

- def **depends_on**

  Abstract method: The implementation should return a list of the components of uncertainty, that this component depends on.

- def **equal_debug**

  A method that is only used for serialization checking.

- def **exp**

  This method provides the broadcast interface for numpy.exp.

- def **fabs**

  This method provides the broadcast interface for numpy.fabs.

- def **gaussian**

  A factory method, that can be used to create instances of uncertain components. This method returns uncertain inputs that are quantified as a gaussian distribution, centered at value, and having the uncertainty sigma.

- def **get_uncertainty**

  *Abstract method: The implementation should return a numeric value (e.g. float,int,long,or **arithmetic.RationalNumber** (p. 210)) representing the standard uncertainty of this component.*

- def **get_value**

  *Abstract method: The implementation should return a numeric value (e.g. float,int,long,or **arithmetic.RationalNumber** (p. 210)) representing the value assigned to the component of uncertainty.*

- def **hypot**

  This method provides an interface for numpy.hypot.

- def **log**

  This method provides the broadcast interface for numpy.log.

- def **log10**

  This method provides the broadcast interface for numpy.log10.

- def **log2**

  This method provides the broadcast interface for numpy.log2.

- def **set_context**

  Assign a context to this component. This method is only used in combination with __str__. If a context is assigned to the instance, the correlation coefficients will be considered for __str__. Otherwise __str__ assumes that there is no correlation among the inputs.

- def **sin**

  This method provides the broadcast interface for numpy.sin.

- def **sinh**

  This method provides the broadcast interface for numpy.sinh.

- def **sqrt**

  This method provides the broadcast interface for numpy.sqrt.

- def **square**

  This method provides the broadcast interface for numpy.sqrt.

- def **tan**

  This method provides the broadcast interface for numpy.tan.

- def **tanh**

  This method provides the broadcast interface for numpy.tanh.

- def **triangular**

  A factory method, that can be used to create instances of uncertain components. This method returns uncertain inputs that are quantified as a triangular distribution, centered at x, and having the half-width a.

- def **uniform**

  A factory method, that can be used to create instances of uncertain components. This method returns uncertain inputs that are quantified as a uniform distribution, centered at x, and having the half-width a.

- def **value_of**

  A factory method, that can be used to create instances of uncertain components. This method returns instances of UncertainNumber depending on the argument.

Static Public Attributes

- tuple **arcsine** = staticmethod(**arcsine**)
- tuple **beta** = staticmethod(**beta**)
- tuple **gaussian** = staticmethod(**gaussian**)
- tuple **triangular** = staticmethod(**triangular**)
- tuple **uniform** = staticmethod(**uniform**)
- tuple **value_of** = staticmethod(**value_of**)

Private Attributes

- **__context**

4.87.2 Member Function Documentation

4.87.2.1 def __abs__ (*self*)

This method returs the absolute value of this instance.

Parameters:

self

4.87.2.2 def __add__ (*self*, *other*)

This method adds the argument to this instance.

Note:

If the argument is not an instance of **UncertainComponent** (p. 289) it will be converted using **UncertainComponent.value_of** (p. 309).

Parameters:

self

other A numeric value.

See also:

UncertainComponent.value_of (p. 309)

4.87.2.3 def __coerce__ (*self*, *other*)

Implementation of coercion rules.

See also:

Coercion - The page describing the coercion rules.

4.87.2.4 def __div__ (*self*, *other*)

This method divides this instance by the argument.

Note:

If the argument is not an instance of **UncertainComponent** (p. 289) it will be converted using **UncertainComponent.value_of** (p. 309).

Parameters:

self

other A numeric value.

See also:

UncertainComponent.value_of (p. 309)

4.87.2.5 def __eq__ (*self*, *other*)

This method is an alias for (self is other). It checks if the argument is identical with the current instance.

Note:

This behavior is enforced to handle special cases. Imagine you want to compare $sin(a \pm u_a) \times (a \pm u_a)$ with $sin(a \pm u_a) \times (b \pm u_b)$ with $a = b; u_a = u_b$. Since in the first case the values are identical, they are dependent. In the second case the values are the same, but we do not know about their independence. Therefore the second case needs a different handling. In order not to confuse these two cases, this method has to check for identity.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the argument is identical to the current instance.

4.87.2.6 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented in **UncertainInput** (p. 311), **BinaryOperation** (p. 88), and **UnaryOperation** (p. 287).

4.87.2.7 def __init__ (*self*)

Default constructor.

Parameters:

self

4.87.2.8 def __invert__ (*self*)

Inverts this instance.

Parameters:

self

4.87.2.9 def __mul__ (*self*, *other*)

This method multiplies the argument by this instance.

Note:

If the argument is not an instance of **UncertainComponent** (p. 289) it will be converted using **UncertainComponent.value_of** (p. 309).

Parameters:

self

other A numeric value.

See also:

UncertainComponent.value_of (p. 309)

4.87.2.10 def __ne__ (*self*, *other*)

This method is an alias for not(self is other). It checks if the argument is not identical with the current instance.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the argument is not identical to the current instance.

See also:

UncertainComponent.__eq__ (p. 295)

4.87.2.11 def __neg__ (*self*)

This method negates this instance.

Parameters:

self

4.87.2.12 def __pow__ (*self*, *other*)

This method raises this to the power of the argument.

Note:

If the argument is not an instance of **UncertainComponent** (p. 289) it will be converted using **UncertainComponent.value_of** (p. 309).

Parameters:

self

other A numeric value.

See also:

UncertainComponent.value_of (p. 309)

4.87.2.13 def __radd__ (*self*, *other*)

This method adds this instance to the argument.

Note:

If the argument is not an instance of **UncertainComponent** (p. 289) it will be converted using **UncertainComponent.value_of** (p. 309).

Parameters:

self

other A numeric value.

See also:

UncertainComponent.value_of (p. 309)

4.87.2.14 def __rdiv__ (*self*, *other*)

This method divides the argument by this instance.

Note:

If the argument is not an instance of **UncertainComponent** (p. 289) it will be converted using **UncertainComponent.value_of** (p. 309).

Parameters:

self

other A numeric value.

See also:

UncertainComponent.value_of (p. 309)

4.87.2.15 def __rmul__ (*self*, *other*)

This method multiplies the argument by this instance.

Note:

If the argument is not an instance of **UncertainComponent** (p. 289) it will be converted using **UncertainComponent.value_of** (p. 309).

Parameters:

self

other A numeric value.

See also:

UncertainComponent.value_of (p. 309)

4.87.2.16 def __rpow__ (*self*, *other*)

This method raises the argument to the power of this instance.

Note:

If the argument is not an instance of **UncertainComponent** (p. 289) it will be converted using **UncertainComponent.value_of** (p. 309).

Parameters:

self

other A numeric value.

See also:

UncertainComponent.value_of (p. 309)

4.87.2.17 def __rsub__ (*self*, *other*)

This method substracts this instance from the argument.

Note:

If the argument is not an instance of **UncertainComponent** (p. 289) it will be converted using **UncertainComponent.value_of** (p. 309).

Parameters:

self

other A numeric value.

See also:

UncertainComponent.value_of (p. 309)

4.87.2.18 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

Reimplemented in **UncertainInput** (p. 312), **BinaryOperation** (p. 89), and **UnaryOperation** (p. 288).

4.87.2.19 def __str__ (*self*)

This method returs the absolute value of this instance.

Parameters:

self

Returns:

A string of the form "<value> +- <uncertainty>" or "<value> +- <uncertainty> [NC]", if no context was provided.

See also:

set_context (p. 306)

4.87.2.20 def __sub__ (*self*, *other*)

This method substracts the argument from this instance.

Note:

If the argument is not an instance of **UncertainComponent** (p. 289) it will be converted using **UncertainComponent.value_of** (p. 309).

Parameters:

self

other A numeric value.

See also:

UncertainComponent.value_of (p. 309)

4.87.2.21 def arccos (*self*)

This method provides the broadcast interface for numpy.arccos.

Parameters:

self

Returns:

The inverse Cosine of this component.

4.87.2.22 def arccosh (*self*)

This method provides the broadcast interface for numpy.arccosh.

Parameters:

self

Returns:

The inverse hyperbolic Cosine of this component.

4.87.2.23 def arcsin (*self*)

This method provides the broadcast interface for numpy.arcsin.

Parameters:

self

Returns:

The inverse Sine of this component.

4.87.2.24 def arcsine (*value*, *dof* = arithmetic.INFINITY)

A factory method, that can be used to create instances of uncertain components. This method returns uncertain inputs that are quantified as an arcsin distribution.

Returns:

An uncertain component.

4.87.2.25 def arcsinh (*self*)

This method provides the broadcast interface for numpy.arcsinh.

Parameters:

self

Returns:

The inverse hyperbolic Sine of this component.

4.87.2.26 def arctan (*self*)

This method provides the broadcast interface for numpy.arctan.

Parameters:

self

Returns:

The inverse Tangent of this component.

4.87.2.27 def arctan2 (*self*, *other*)

This method provides an interface for numpy.arctan2.

Parameters:

self

other A numeric value.

See also:

UncertainComponent.value_of (p. 309)

4.87.2.28 def arctanh (*self*)

This method provides the broadcast interface for numpy.arctanh.

Parameters:

self

Returns:

The inverse hyperbolic Tangent of this component.

4.87.2.29 def arithmetic_check (*self*)

This method checks this instance for mathematical correctness. You should overload this method, if your class is not defined for specific argument values. If any (mathematical) invalid values have been assigned, your implementation should raise an ArithmeticError explaining the problem. This method is usually called within the constructor of a class, after the members have been initialized.

Parameters:

self

Reimplemented in **Div** (p. 144), **Sqrt** (p. 240), **Log** (p. 154), **ArcSin** (p. 68), **ArcCos** (p. 63), **ArcCosh** (p. 65), and **ArcTanh** (p. 81).

4.87.2.30 def beta (*value*, *p*, *q*, *dof* = arithmetic.INFINITY)

A factory method, that can be used to create instances of uncertain components. This method returns uncertain inputs that are quantified as a beta distribution, having the parameters p and q.

Parameters:

value The assigned value.

p A numeric value, representing p.

q A numeric value, representing q

dof The assigned number of degrees of freedom.

Returns:

An uncertain component.

4.87.2.31 def cos (*self*)

This method provides the broadcast interface for numpy.cos.

Parameters:

self

Returns:

The Cosine of this component.

4.87.2.32 def cosh (*self*)

This method provides the broadcast interface for numpy.cosh.

Parameters:

self

Returns:

The hyperbolic Cosine of this component.

4.87.2.33 def depends_on (*self*)

Abstract method: The implementation should return a list of the components of uncertainty, that this component depends on.

Returns:

A list of the components of uncertainty.

Reimplemented in **UncertainInput** (p. 312), **BinaryOperation** (p. 89), and **UnaryOperation** (p. 288).

4.87.2.34 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has equal attributes as the argument

Reimplemented in **UncertainInput** (p. 312), **BinaryOperation** (p. 89), **UnaryOperation** (p. 288), **Add** (p. 51), **ArcTan2** (p. 79), **Mul** (p. 162), **Div** (p. 144), **Sub** (p. 243), **Pow** (p. 174), **Cos** (p. 110), **Sin** (p. 233), **Tan** (p. 246), **Sqrt** (p. 241), **Log** (p. 154), **ArcSin** (p. 69), **ArcSinh** (p. 73), **ArcCos** (p. 63), **ArcCosh** (p. 65), **ArcTan** (p. 75), **ArcTanh** (p. 81), **Cosh** (p. 114), **Sinh** (p. 237), **Tanh** (p. 249), **Exp** (p. 146), **Abs** (p. 48), and **Neg** (p. 169).

4.87.2.35 def exp (*self*)

This method provides the broadcast interface for numpy.exp.

Parameters:

self

Returns:

The Exponential of this component.

4.87.2.36 def fabs (*self*)

This method provides the broadcast interface for numpy.fabs.

Parameters:

self

Returns:

The absolute value of this component.

4.87.2.37 def gaussian (*value*, *sigma*, *dof* = arithmetic.INFINITY)

A factory method, that can be used to create instances of uncertain components. This method returns uncertain inputs that are quantified as a gaussian distribution, centered at value, and having the uncertainty sigma.

Parameters:

value A numeric value, representing x.

sigma A numeric value, representing a

dof The assigned number of degrees of freedom.

Returns:

An uncertain component.

4.87.2.38 def get_uncertainty (*self*, *component*)

Abstract method: The implementation should return a numeric value (e.g. float,int,long,or **arithmetic.RationalNumber** (p. 210)) representing the standard uncertainty of this component.

Parameters:

self

component Another instance of uncertainty. If the argument is this instance the uncertainty is returned, $\frac{0}{1}$ should be returned otherwise. This is analogous to taking the derivate of an independent variable. For further explanation see "The GUM tree".

Returns:

A numeric value, representing the standard uncertainty.

See also:

arithmetic.RationalNumber (p. 210)
"The "GUM Tree": A software design pattern for handling measurement uncertainty"; B. D. Hall; Industrial Research Report 1291; Measurements Standards Laboratory New Zealand (2003).

Reimplemented in **UncertainInput** (p. 313), **Add** (p. 51), **ArcTan2** (p. 79), **Mul** (p. 163), **Div** (p. 144), **Sub** (p. 243), **Pow** (p. 175), **Cos** (p. 110), **Sin** (p. 233), **Tan** (p. 246), **Sqrt** (p. 241), **Log** (p. 154), **ArcSin** (p. 69), **ArcSinh** (p. 73), **ArcCos** (p. 63), **ArcCosh** (p. 66), **ArcTan** (p. 75), **ArcTanh** (p. 82), **Cosh** (p. 114), **Sinh** (p. 238), **Tanh** (p. 249), **Exp** (p. 146), **Abs** (p. 48), and **Neg** (p. 169).

4.87.2.39 def get_value (*self*)

Abstract method: The implementation should return a numeric value (e.g. float,int,long,or **arithmetic.RationalNumber** (p. 210)) representing the value assigned to the component of uncertainty.

Parameters:

self

Returns:

A numeric value, representing the value.

Reimplemented in **UncertainInput** (p. 313), **Add** (p. 51), **ArcTan2** (p. 80), **Mul** (p. 163), **Div** (p. 145), **Sub** (p. 243), **Pow** (p. 175), **Cos** (p. 110), **Sin** (p. 234), **Tan** (p. 246), **Sqrt** (p. 241), **Log** (p. 155), **ArcSin** (p. 69), **ArcSinh** (p. 74), **ArcCos** (p. 64), **ArcCosh** (p. 66), **ArcTan** (p. 75), **ArcTanh** (p. 82), **Cosh** (p. 115), **Sinh** (p. 238), **Tanh** (p. 249), **Exp** (p. 147), **Abs** (p. 48), and **Neg** (p. 169).

4.87.2.40 def hypot (*self*, *other*)

This method provides an interface for numpy.hypot.

Parameters:

self

other A numeric value.

See also:

UncertainComponent.value_of (p. 309)

4.87.2.41 def log (*self*)

This method provides the broadcast interface for numpy.log.

Parameters:

self

Returns:

The Natural Logarithm of this component.

4.87.2.42 def log10 (*self*)

This method provides the broadcast interface for numpy.log10.

Parameters:

self

Returns:

The decadic Logarithm of this component.

4.87.2.43 def log2 (*self*)

This method provides the broadcast interface for numpy.log2.

Parameters:

self

Returns:

The decadic Logarithm of this component.

4.87.2.44 def set_context (*self*, *context*)

Assign a context to this component. This method is only used in combination with __str__. If a context is assigned to the instance, the correlation coefficients will be considered for __str__. Otherwise __str__ assumes that there is no correlation among the inputs.

Parameters:

context An instance of **Context** (p. 101).

self

See also:

__str__ (p. 299)
Context (p. 101)

4.87.2.45 def sin (*self*)

This method provides the broadcast interface for numpy.sin.

Parameters:

self

Returns:

The Sine of this component.

4.87.2.46 def sinh (*self*)

This method provides the broadcast interface for numpy.sinh.

Parameters:

self

Returns:

The hyperbolic Sine of this component.

4.87.2.47 def sqrt (*self*)

This method provides the broadcast interface for numpy.sqrt.

Parameters:

self

Returns:

The Square Root of this component.

4.87.2.48 def square (*self*)

This method provides the broadcast interface for numpy.sqrt.

Parameters:

self

Returns:

The Square Root of this component.

4.87.2.49 def tan (*self*)

This method provides the broadcast interface for numpy.tan.

Parameters:

self

Returns:

The Tangent of this component.

4.87.2.50 def tanh (*self*)

This method provides the broadcast interface for numpy.tanh.

Parameters:

self

Returns:

The hyperbolic Tangent of this component.

4.87.2.51 def triangular (*value*, *halfwitdh*, *dof* = arithmetic.INFINITY)

A factory method, that can be used to create instances of uncertain components. This method returns uncertain inputs that are quantified as a triangular distribution, centered at x, and having the half-width a.

Parameters:

value A numeric value, representing x.

halfwitdh A numeric value, representing a

dof The assigned number of degrees of freedom.

Returns:

An uncertain component, having the uncertainty $u(x) = \frac{a}{\sqrt{6}}$.

4.87.2.52 def uniform (*value*, *halfwitdh*, *dof* = arithmetic.INFINITY)

A factory method, that can be used to create instances of uncertain components. This method returns uncertain inputs that are quantified as a uniform distribution, centered at x, and having the half-width a.

Parameters:

value A numeric value, representing x.

halfwitdh A numeric value, representing a.

dof The assigned number of degrees of freedom.

Returns:

An uncertain component, having the uncertainty $u(x) = \frac{a}{\sqrt{3}}$.

4.87.2.53 def value_of (*value*)

A factory method, that can be used to create instances of uncertain components. This method returns instances of UncertainNumber depending on the argument.

Parameters:

value An instance of UncertainNumber or a numeric value.

Returns:

The argument, if it is already an instance of UncertainNumber, or a new instance of **UncertainInput** (p. 309) (having an uncertainty of 0.0) if the argument is a numeric value (i.e. int,float...).

Exceptions:

TypeError If the argument is a quantity. You cannot encapsulate quantites in UncertainValues. Plase use Quantity(UncertainValue) instead.

4.87.3 Member Data Documentation

4.87.3.1 __context [private]

4.87.3.2 tuple arcsine = staticmethod(arcsine) [static]

4.87.3.3 tuple beta = staticmethod(beta) [static]

4.87.3.4 tuple gaussian = staticmethod(gaussian) [static]

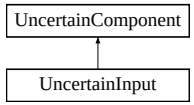
4.87.3.5 tuple triangular = staticmethod(triangular) [static]

4.87.3.6 tuple uniform = staticmethod(uniform) [static]

4.87.3.7 tuple value_of = staticmethod(value_of) [static]

4.88 UncertainInput Class Reference

Inheritance diagram for UncertainInput::



4.88.1 Detailed Description

This class provides the model for uncertain inputs, that are referred to as "Leafs" in "The GUM tree".

See also:

"The "GUM Tree": A software design pattern for handling measurement uncertainty"; B. D. Hall; Industrial Research Report 1291; Measurements Standards Laboratory New Zealand (2003).

Public Member Functions

- def **__getstate__**
  Serialization using pickle.
- def **__hash__**
  Hash this instance.
- def **__init__**
  Default constructor.
- def **__setstate__**
  Deserialization using pickle.
- def **depends_on**
  Returns a list containing this element.
- def **equal_debug**
  A method that is only used for serialization checking.
- def **get_dof**
  Returns the assigned degrees of freedom.
- def **get_uncertainty**
  Returns the assigned uncertainty.
- def **get_value**
  Returns the assigned value.

Private Attributes

- **__dof**
- **__uncertainty**
- **__value**

Static Private Attributes

- float **__dof** = 0.0
- float **__uncertainty** = 0.0
- float **__value** = 0.0

4.88.2 Member Function Documentation

4.88.2.1 def __getstate__ (*self*)

Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented from **UncertainComponent** (p. 295).

4.88.2.2 def __hash__ (*self*)

Hash this instance.

4.88.2.3 def __init__ (*self*, *value*, *uncertainty*, *dof* = **arithmetic.INFINITY**)

Default constructor.

Note:

The parameters of the input must not be instances of **UncertainComponent** (p. 289) nor **quantities.Quantity** (p. 184). Create quantities, having an **UncertainInput** (p. 309) as numeric argument instead.

Parameters:

self

value The numeric value of the input.

dof The assigned component of degrees of freedom.

uncertainty The standard uncertainty of the input.

See also:

UncertainQuantity.py

4.88.2.4 def __setstate__ (*self*, *state*)

Deserialization using pickle.

Parameters:

self

state The state of the object.

Reimplemented from **UncertainComponent** (p. 299).

4.88.2.5 def depends_on (*self*)

Returns a list containing this element.

Returns:

A list of the components of uncertainty.

Reimplemented from **UncertainComponent** (p. 303).

4.88.2.6 def equal_debug (*self*, *other*)

A method that is only used for serialization checking.

Parameters:

self

other Another instance of **UncertainComponent** (p. 289)

Returns:

True, if the instance has the same attribute values as the argument

Reimplemented from **UncertainComponent** (p. 303).

4.88.2.7 def get_dof (*self*)

Returns the assigned degrees of freedom.

Parameters:

self

Returns:

A numeric value or **arithmetic.INFINITY** (p. 4), representing the value.

4.88.2.8 def get_uncertainty (*self*, *component*)

Returns the assigned uncertainty.

Parameters:

self

component Another component of uncertainty.

Returns:

A numeric value, representing the standard uncertainty.

See also:

UncertainComponent.get_uncertainty (p. 304)

Reimplemented from **UncertainComponent** (p. 304).

4.88.2.9 def get_value (*self*)

Returns the assigned value.

Parameters:

self

Returns:

A numeric value, representing the value.

Reimplemented from **UncertainComponent** (p. 305).

4.88.3 Member Data Documentation

4.88.3.1 __dof [private]

4.88.3.2 float __dof = 0.0 [static, private]

4.88.3.3 __uncertainty [private]

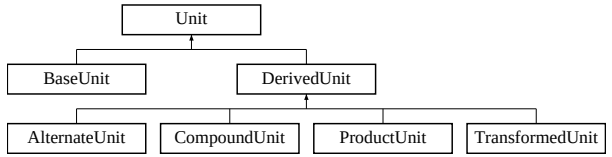
4.88.3.4 float __uncertainty = 0.0 [static, private]

4.88.3.5 __value [private]

4.88.3.6 float __value = 0.0 [static, private]

4.89 Unit Class Reference

Inheritance diagram for Unit::



4.89.1 Detailed Description

An abstract class to model physical units.

This class provides an interface to model physical units.

Attention:

You have to use one of its silblings to get any effect.

Public Member Functions

- def **__add__**

  *Support of Adding an offset to the current unit (i.e. $Celsus = Kelvin + 253.15$). This function returns a **TransformedUnit** (p. 281) that represents adding the offset to the current unit.*

- def **__coerce__**

  Implementation of coercion rules. This implementation ensures that transformed units can be created from units.

- def **__div__**

  *Support for dividing units and numeric values. This function returns a new **Unit** (p. 314) that represents the operation.*

- def **__eq__**

  Check for if two units are equal.

- def **__getstate__**

  Abstract method: Serialization using pickle.

- def **__invert__**

  *Support inversion of units. This function returns a new **Unit** (p. 314) that represents the operation. Suppose your unit is $[U]$, then the inverted unit is $[\frac{1}{U}]$.*

- def **__mul__**

  *Suport for multiplying a numeric value or unit. This function returns a new **Unit** (p. 314) that represents multiplying the factor or unit to the current unit.*

- def **__ne__**

  Check for if two units are unequal.

- def **__pow__**

  *Support integer powers. This function returns a new **Unit** (p. 314) that represents the power of the current unit.*

- def **__setstate__**

  Abstract method: Deserialization using pickle.

- def **__str__**

  *Support of printing units. The silblings of unit override this method. It should print the symbols of the units that form the unit. For example a **ProductUnit** (p. 176) might print* `kg*m*s^(-2)`.

- def **__sub__**

  *Support for Substracting an offset. This function works in a similar way as **Unit.__add__** (p. 316).*

- def **compound**

  Support compound units. This method returns a compound unit of the current unit and the argument. Both units have to describe the same physical dimension.

- def **get_dimension**

  Get the corresponding physical dimension.

- def **get_operator_to**

  Convert units. This method returns an operator that converts values that have been formed with the current unit to another other unit.

- def **get_system_unit**

  Get the corresponding system unit. The physical model is used to determine the mapping to the system unit.

- def **is_compatible**

  Check if two units can be converted to each other.

- def **root**

  *Support of integer roots. This function returns a new **Unit** (p. 314) that represents the root of the current unit.*

- def **sqrt**

  *Support of square root. This function returns a new **Unit** (p. 314) that represents the square root of the current unit.*

- def **to_system_unit**

  Abstract Function to convert to corresponding system unit.

Private Member Functions

- def **__getTransformOf**

  Helper function to get the transformation to the system unit. This method returns an operator that converts values that have been formed with the current unit to its system unit.

- def **__rootInstance**

  Helper function to get the n-th root instance of the unit.

- def **__transformUnit**

  Helper function that applies a transformation to the current unit.

4.89.2 Member Function Documentation

4.89.2.1 def __add__ (*self*, *other*)

Support of Adding an offset to the current unit (i.e. $Celsus = Kelvin + 253.15$). This function returns a **TransformedUnit** (p. 281) that represents adding the offset to the current unit.

Parameters:

self

other A numerical value to add to the unit.

Returns:

A transformed unit that represents adding the offset to the current unit.

See also:

TransformedUnit (p. 281)

4.89.2.2 def __coerce__ (*self*, *other*)

Implementation of coercion rules. This implementation ensures that transformed units can be created from units.

4.89.2.3 def __div__ (*self*, *other*)

Support for dividing units and numeric values. This function returns a new **Unit** (p. 314) that represents the operation.

Parameters:

self

other A number or unit.

Exceptions:

ZeroDivisionError This error is raised if the divisor is zero.

Returns:

A new unit representing the operation.

See also:

ProductUnit (p. 176)
TransformedUnit (p. 281)

Reimplemented in **ProductUnit** (p. 179).

4.89.2.4 def __eq__ (*self*, *other*)

Check for if two units are equal.

Parameters:

self

other Another instance of a **Unit** (p. 314) or its subclasses.

Returns:

True, if this unit and the argument are equal.

Reimplemented in **BaseUnit** (p. 85), **AlternateUnit** (p. 57), **CompoundUnit** (p. 97), **ProductUnit** (p. 179), and **TransformedUnit** (p. 283).

4.89.2.5 def __getstate__ (*self*)

Abstract method: Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented in **BaseUnit** (p. 85), **AlternateUnit** (p. 58), **CompoundUnit** (p. 98), **ProductUnit** (p. 180), and **TransformedUnit** (p. 283).

4.89.2.6 def __getTransformOf (*self*) [private]

Helper function to get the transformation to the system unit. This method returns an operator that converts values that have been formed with the current unit to its system unit.

Parameters:

self

Returns:

A converter to the system unit.

Exceptions:

qexceptions.ConversionException (p. ***108***) If a conversion is not possible an exception is raised (e.g. if the unit has a fractional exponent, or if the transformation is not linear).

See also:

operators (p. 28)

4.89.2.7 def __invert__ (*self*)

Support inversion of units. This function returns a new **Unit** (p. 314) that represents the operation. Suppose your unit is $[U]$, then the inverted unit is $[\frac{1}{U}]$.

Returns:

A new unit representing the operation.

See also:

ProductUnit (p. 176)

4.89.2.8 def __mul__ (*self*, *other*)

Suport for multiplying a numeric value or unit. This function returns a new **Unit** (p. 314) that represents multiplying the factor or unit to the current unit.

Parameters:

self

other A number or unit.

Returns:

A new unit representing the operation.

See also:

TransformedUnit (p. 281)
ProductUnit (p. 176)
ONE (p. 23)

4.89.2.9 def __ne__ (*self*, *other*)

Check for if two units are unequal.

Parameters:

self

other Another instance of a **Unit** (p. 314) or its subclasses.

Returns:

True, if this unit and the argument are unequal.

4.89.2.10 def __pow__ (*self*, *other*)

Support integer powers. This function returns a new **Unit** (p. 314) that represents the power of the current unit.

Attention:

In order to preserve dimensional consistency we do only allow integers or rational numbers as powers of units.

Parameters:

self

other An integer.

Returns:

A new unit representing the operation.

See also:

ProductUnit (p. 176)

4.89.2.11 def __rootInstance (*self*, *unit*, *root*) [private]

Helper function to get the n-th root instance of the unit.

Parameters:

self

unit The unit to be rooted.

root An integer to be used as root.

Returns:

A new unit representing the operation.

See also:

ProductUnit (p. 176)

4.89.2.12 def __setstate__ (*self*, *state*)

Abstract method: Deserialization using pickle.

Parameters:

self

state The state of the object.

Reimplemented in **BaseUnit** (p. 86), **AlternateUnit** (p. 58), **CompoundUnit** (p. 98), **ProductUnit** (p. 180), and **TransformedUnit** (p. 284).

4.89.2.13 def __str__ (*self*)

Support of printing units. The silblings of unit override this method. It should print the symbols of the units that form the unit. For example a **ProductUnit** (p. 176) might print `kg*m*s^(-2)`.

Parameters:

self

Returns:

A string describing this unit.

Attention:

The subclasses override this method, calling **Unit.__str__** (p. 320) has no effect.

Reimplemented in **BaseUnit** (p. 86), **AlternateUnit** (p. 59), **CompoundUnit** (p. 98), **ProductUnit** (p. 181), and **TransformedUnit** (p. 284).

4.89.2.14 def __sub__ (*self*, *other*)

Support for Substracting an offset. This function works in a similar way as **Unit.__add__** (p. 316).

Parameters:

self

other A numerical value to substract from the unit.

Returns:

A transformed unit that represents substracting the offset from the current unit.

See also:

TransformedUnit (p. 281)
Unit.__add__ (p. 316)

4.89.2.15 def __transformUnit (*self*, *operation*) [private]

Helper function that applies a transformation to the current unit.

Parameters:

self

operation The operation to be performed.

Returns:

A new unit representing the current unit after applying the operation.

4.89.2.16 def compound (*self*, *other*)

Support compound units. This method returns a compound unit of the current unit and the argument. Both units have to describe the same physical dimension.

Parameters:

self

other Another unit describing the same dimension.

Returns:

A new compound unit.

Exceptions:

TypeError If the units describe different dimensions.

4.89.2.17 def get_dimension (*self*)

Get the corresponding physical dimension.

Parameters:

self

Returns:

The corresponding physical dimension.

Attention:

This method is intended to be final for the predefined silblings of **Unit** (p. 314). You only have to override it if you are directly inheriting from **Unit** (p. 314).

4.89.2.18 def get_operator_to (*self*, *unit*)

Convert units. This method returns an operator that converts values that have been formed with the current unit to another other unit.

Parameters:

self

unit The unit to convert to.

Returns:

A converter to the argument.

Exceptions:

qexceptions.ConversionException *(*p. ***108)*** If a conversion is not possible an exception is raised (i.e. if the units describe different physical dimensions).

See also:

operators (p. 28)

4.89.2.19 def get_system_unit (*self*)

Get the corresponding system unit. The physical model is used to determine the mapping to the system unit.

Parameters:

self

Returns:

The system unit.

See also:

PhysicalModel (p. 172)

Attention:

The subclasses override this method, calling **Unit.get_system_unit** (p. 322) has no effect.

Reimplemented in **BaseUnit** (p. 87), **AlternateUnit** (p. 60), **CompoundUnit** (p. 99), **ProductUnit** (p. 181), and **TransformedUnit** (p. 284).

4.89.2.20 def is_compatible (*self*, *other*)

Check if two units can be converted to each other.

Two units are compatible if their corresponding system units match, or if both units describe the same physical dimension.

Parameters:

self

other Another unit to compare to.

Returns:

True, if the units are compatible.

4.89.2.21 def root (*self*, *other*)

Support of integer roots. This function returns a new **Unit** (p. 314) that represents the root of the current unit.

Parameters:

self

other An integer.

Exceptions:

ArithmeticError If the root is zero this function fails.

Returns:

A new unit representing the operation.

See also:

ProductUnit (p. 176)

4.89.2.22 def sqrt (*self*)

Support of square root. This function returns a new **Unit** (p. 314) that represents the square root of the current unit.

Parameters:

self

Returns:

A new unit representing the operation.

See also:

ProductUnit (p. 176)
root (p. 323)

4.89.2.23 def to_system_unit (*self*)

Abstract Function to convert to corresponding system unit.

Parameters:

self

Returns:

The corresponding system unit.

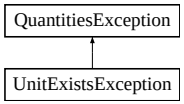
Attention:

The subclasses override this method, calling **Unit.to_system_unit** (p. 324) has no effect.

Reimplemented in **BaseUnit** (p. 87), **AlternateUnit** (p. 60), **CompoundUnit** (p. 99), **ProductUnit** (p. 183), and **TransformedUnit** (p. 285).

4.90 UnitExistsException Class Reference

Inheritance diagram for UnitExistsException::



4.90.1 Detailed Description

Exception that is raised when a dimension, base unit, or alternate unit of the same type has already been created.

See also:

units.BaseUnit (p. 83)
units.AlternateUnit (p. 56)
units.Dimension (p. 138)

Public Member Functions

- def **__init__**
  Default constructor.
- def **__str__**
  Returns a string describing this exception.

Private Attributes

- **__unit__**

4.90.2 Member Function Documentation

4.90.2.1 def __init__ (*self*, *unit*, *args*)

Default constructor.

Parameters:

self
unit The unit that raised this exception.
args Additional arguments of this exception.

4.90.2.2 def __str__ (*self*)

Returns a string describing this exception.

Parameters:

self

Returns:

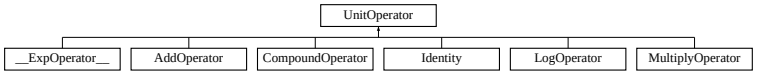
A string that describes this exception.

4.90.3 Member Data Documentation

4.90.3.1 __unit__ [private]

4.91 UnitOperator Class Reference

Inheritance diagram for UnitOperator::



4.91.1 Detailed Description

Basic abstract Operator to use on units.

Attention:

This class is intended to be abstract. You have to use one of its silblings get any effect.

Public Member Functions

- def **__eq__**
  Test for equality.
- def **__getstate__**
  Abstract method: Serialization using pickle.
- def **__invert__**
  Invert this operation.
- def **__mul__**
  Perform the current operation on another operator.
- def **__setstate__**
  Abstract method: Deserialization using pickle.
- def **__str__**
  Represent this operation by a string.
- def **convert**
  Convert a value.
- def **is_linear**
  Check if the operator is linear.

4.91.2 Member Function Documentation

4.91.2.1 def __eq__ (*self*, *other*)

Test for equality.

Parameters:

self

other Another **UnitOperator** (p. 326).

Reimplemented in **__ExpOperator__** (p. 39), **LogOperator** (p. 158), **AddOperator** (p. 53), **MultiplyOperator** (p. 165), **CompoundOperator** (p. 93), and **Identity** (p. 149).

4.91.2.2 def __getstate__ (*self*)

Abstract method: Serialization using pickle.

Parameters:

self

Returns:

A string that represents the serialized form of this instance.

Reimplemented in **__ExpOperator__** (p. 39), **LogOperator** (p. 158), **AddOperator** (p. 53), **MultiplyOperator** (p. 165), **CompoundOperator** (p. 94), and **Identity** (p. 149).

4.91.2.3 def __invert__ (*self*)

Invert this operation.

This method returns a new operator that represents the inversion of this operator.

Attention:

This method is intended to be abstract. The silblings of this class override it in order to get an effect.

Parameters:

self

Returns:

The inverse operation of this operation.

Reimplemented in **__ExpOperator__** (p. 39), **LogOperator** (p. 158), **AddOperator** (p. 54), **MultiplyOperator** (p. 165), **CompoundOperator** (p. 94), and **Identity** (p. 150).

4.91.2.4 def __mul__ (*self*, *otherOperator*)

Perform the current operation on another operator.

Another operation $g(x)$ will be performed on this operator $f(x)$. So that the new Operator is $f \times g = g(f(x))$.

Parameters:

self

otherOperator The other operator to concat.

Returns:

The resulting operator.

Reimplemented in **AddOperator** (p. 54), **MultiplyOperator** (p. 166), and **Identity** (p. 150).

4.91.2.5 def __setstate__ (*self*, *state*)

Abstract method: Deserialization using pickle.

Parameters:

self

state The state of the object.

Reimplemented in **__ExpOperator__** (p. 40), **LogOperator** (p. 159), **AddOperator** (p. 54), **MultiplyOperator** (p. 166), **CompoundOperator** (p. 94), and **Identity** (p. 150).

4.91.2.6 def __str__ (*self*)

Represent this operation by a string.

Attention:

This method is intended to be abstract. The silblings of this class override it in order to get an effect.

Parameters:

self

Returns:

A string describing this operation.

Reimplemented in **__ExpOperator__** (p. 40), **LogOperator** (p. 159), **AddOperator** (p. 55), **MultiplyOperator** (p. 167), **CompoundOperator** (p. 95), and **Identity** (p. 151).

4.91.2.7 def convert (*self*, *value*)

Convert a value.

This method performs the desired operation on an absolute value.

Attention:

This method is intended to be abstract. The silblings of this class override it in order to get an effect.

Parameters:

self

value The value to convert.

Returns:

The converted value

Reimplemented in **__ExpOperator__** (p. 40), **LogOperator** (p. 159), **AddOperator** (p. 55), **MultiplyOperator** (p. 167), **CompoundOperator** (p. 95), and **Identity** (p. 151).

4.91.2.8 def is_linear (*self*)

Check if the operator is linear.

This method checks if this operator is linear or not.

Attention:

This method is intended to be abstract. The silblings of this class override it in order to get an effect.

Parameters:

self

Returns:

True, if the Operator is linear.

Reimplemented in **__ExpOperator__** (p. 41), **LogOperator** (p. 160), **AddOperator** (p. 56), **MultiplyOperator** (p. 167), **CompoundOperator** (p. 95), and **Identity** (p. 151).

4.92 UnitsManager Class Reference

4.92.1 Detailed Description

This manages the alternate and base units as well as the physical dimensions.

See also:

AlternateUnit (p. 56)
BaseUnit (p. 83)
Dimension (p. 138)

Public Member Functions

- def **__init__**
  This is the default constructor.
- def **addUnit**
  This is a helper function to add the units to the dictionary.
- def **existsUnit**
  *Check if a **BaseUnit** (p. 83), **AlternateUnit** (p. 56) or **Dimension** (p. 138) is already contained.*
- def **get_model**
  Return the global physical model used.
- def **set_model**
  Set the global physical model to be used.

Static Private Attributes

- **__physicalModel** = None
  Physical Model used for units.
- dictionary **__unitsDictionary__** = {None:None}

4.92.2 Member Function Documentation

4.92.2.1 def __init__ (*self*)

This is the default constructor.

Parameters:

self

4.92.2.2 def addUnit (*self*, *unit*)

This is a helper function to add the units to the dictionary.

Parameters:

self

unit An instance of an **BaseUnit** (p. 83) or **AlternateUnit** (p. 56) to add.

Exceptions:

UnitExistsException If the same symbol already exists in the dictionary of units.

4.92.2.3 def existsUnit (*self*, *unit*)

Check if a **BaseUnit** (p. 83), **AlternateUnit** (p. 56) or **Dimension** (p. 138) is already contained.

This function checks only for the existence of a Symbol. So that no Symbols of units and dimensions are defined twice.

Parameters:

self

unit An instance of a **BaseUnit** (p. 83), **AlternateUnit** (p. 56) or **Dimension** (p. 138) to be checked for.

Returns:

True, if the Symbol of the unit/dimension already existed. False, otherwise.

4.92.2.4 def get_model (*self*)

Return the global physical model used.

Parameters:

self

Returns:

The current physical model used.

Attention:

This function returns None, if no model is currently in use.

4.92.2.5 def set_model (*self*, *physicalModel*)

Set the global physical model to be used.

Parameters:

self

physicalModel The model to be used.

4.92.3 Member Data Documentation

4.92.3.1 __physicalModel = None `[static, private]`

Physical Model used for units.

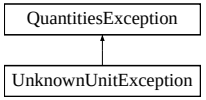
4.92.3.2 dictionary __unitsDictionary__ = {None:None} `[static, private]`

Dictionary of BaseUnits and AlternateUnits created.

It maps the symbol from the respective base or alternate unit to an instance of the **BaseUnit** (p. 83) created.

4.93 UnknownUnitException Class Reference

Inheritance diagram for UnknownUnitException::



4.93.1 Detailed Description

An exception that is raised whenever an unexpected unit was used.

See also:

si.SIModel.get_dimension (p. 232)

Public Member Functions

- def **__init__**
  The default constructor.
- def **__str__**
  Returns a string describing this exception.

Private Attributes

- **__unit__**

4.93.2 Member Function Documentation

4.93.2.1 def __init__ (*self*, *unit*, *args*)

The default constructor.

Parameters:

self

unit An instance of a unit that is unknown.

args Additional arguments of this exception.

4.93.2.2 def __str__ (*self*)

Returns a string describing this exception.

Parameters:

self

Returns:

String that describes the exception.

4.93.3 Member Data Documentation

4.93.3.1 __unit__ [private]

5 SCUQ File Documentation

5.1 __init__.py File Reference

5.1.1 Detailed Description

This file is evaluated whenever the quantities package is loaded.

It loads the modules neccessary for operating this package. It also performs some global initialization.

Author:

Thomas Reidemeister

Namespaces

- namespace **scuq**
- namespace **scuq::__init__**

Variables

- list **__all__** = ["arithmetic", "units", "qexceptions", "si", "quantities", "operators", "ucomponents", "cucomponents"]

  The modules contained within the quantities package.

5.2 arithmetic.py File Reference

5.2.1 Detailed Description

This file contains several functions and classes that are used for numeric computations in the other modules of this library.

Author:

Thomas Reidemeister

Namespaces

- namespace **scuq::arithmetic**

Classes

- class **RationalNumber**

  This class provides support for rational numbers.

Functions

- def **complex_to_matrix**

  This function converts a complex number to a column vector.

- def **gcd**

  Calculate the greatest common divisor.

- def **rational**

  This function provides an interface for rational numbers creation, as suggested in PEP 239.

Variables

- string **INFINITY** = "inf"

  Global constant for infinity that is used in combination with the degrees of freedom evaluation.

5.3 cucomponents.py File Reference

5.3.1 Detailed Description

This file contains the module to model complex uncertain values.

Author:

`Thomas Reidemeister`

Namespaces

- namespace **scuq::cucomponents**

Classes

- class **Abs**
  This class models taking the absolute value of a complex function.
- class **Add**
  This class models adding two complex values.
- class **ArcCos**
  This class models the inverse cosine function.
- class **ArcCosh**
  This class models the inverse hyperbolic cosine function.
- class **ArcSin**
  This class models the inverse sine function.
- class **ArcSinh**
  This class models the inverse hyperbolic sine function.
- class **ArcTan**
  This class models the inverse tangent function.
- class **ArcTan2**
  This class models two-argument inverse tangent.
- class **ArcTanh**
  This class models the inverse hyperbolic tangent function.
- class **CBinaryOperation**
  This abstract class models a binary operation.
- class **Conjugate**

This class models taking the negative of a complex value.

- class **Context**
  This class provides a context for complex-valued uncertainty evaluations. It manages the correlation coefficients and is able to evaluate the effective degrees of freedom.
- class **Cos**
  This class models the cosine function.
- class **Cosh**
  This class models the hyperbolic cosine function.
- class **CUnaryOperation**
  This abstract class models an unary operation.
- class **CUncertainComponent**
  This is the abstract super class of all complex valued uncertain components. Despite defining the interface for complex valued uncertain components, it also provides a set of factory methods that act as an interface for `numpy`.
- class **CUncertainInput**
  This class models a complex-valued input of a function.
- class **Div**
  This class models dividing two complex values.
- class **Exp**
  This class models the exponential function e^x. x denotes the sibling of this instance.
- class **Inv**
  This class models inverting complex values. Let an instance of this class model the complex value x then this class models $\frac{1}{x}$.
- class **Log**
  This class models logarithms having a real base. However, the base cannot be uncertain.
- class **Mul**
  This class models multiplying two complex values.
- class **Neg**
  This class models taking the negative of a complex value.
- class **Pow**
  This class models complex powers.
- class **Sin**

This class models the sine function.

- class **Sinh**
  This class models the hyperbolic sine function.
- class **Sqrt**
  This class models taking the square root of an uncertain component.
- class **Sub**
  This class models taking the difference of two complex values.
- class **Tan**
  This class models the tangent function.
- class **Tanh**
  This class models the hyperbolic tangent function.

Functions

- def **complex_to_matrix**
  This function transforms a complex value into a matrix.

5.4 operators.py File Reference

5.4.1 Detailed Description

This file contains the classes necessary to define, use, and handle operations on units.

Author:

```
Thomas Reidemeister
```

Namespaces

- namespace **scuq::operators**

Classes

- class **__ExpOperator__**
  This class provides an Interface for exponential operators. It is used as helper for the ***LogOperator*** (p. *156*)*.*
- class **AddOperator**
  This class provides an Interface for offset operators.

- class **CompoundOperator**
  Compound Operator.
- class **Identity**
  This class provides an Interface for the identity Operator.
- class **LogOperator**
  This class provides an interface for logarithmic operators.
- class **MultiplyOperator**
  This class provides an Interface for factor operators.
- class **UnitOperator**
  Basic abstract Operator to use on units.

Variables

- tuple **IDENTITY** = Identity()

5.5 qexceptions.py File Reference

5.5.1 Detailed Description

This file contains a variety of exception definitions that are used by the quantities package.

Author:

```
Thomas Reidemeister
```

Namespaces

- namespace **scuq::qexceptions**

Classes

- class **ConversionException**
  General exception that is raised whenever a unit conversion fails.
- class **NotDimensionlessException**
  Exception that is raised whenever a a unit is not dimensionless where it has to be.
- class **QuantitiesException**
  General class for qexceptions of this module.

- class **UnitExistsException**
  Exception that is raised when a dimension, base unit, or alternate unit of the same type has already been created.

- class **UnknownUnitException**
  An exception that is raised whenever an unexpected unit was used.

5.6 quantities.py File Reference

5.6.1 Detailed Description

This file contains the classes to model, handle, and use physical quantities.

It also contains some base quantities that can be used to derive combined quantities.

Author:

`Thomas Reidemeister`

Namespaces

- namespace **scuq::quantities**

Classes

- class **Quantity**
  Base class that provides an interface to model quantities.

Functions

- def **is_strict**
  *An abbreviation for **Quantity.is_strict** (p. 210).*

- def **set_strict**
  *An abbreviation for **Quantity.set_strict** (p. 210).*

5.7 si.py File Reference

5.7.1 Detailed Description

This file contains the predefined SI units.

It models SI base units and SI alternate units. The alternate units have been formed as product other alternate SI units where possible as described in NIST 330.

See also:

"The International System of Units"; Barry N. Taylor; NIST 330 (2001)

Author:

`Thomas Reidemeister`

Namespaces

- namespace **scuq::si**

Classes

- class **SIModel**
  The interface for a physical model for SI units.

Variables

- tuple **__model** = SIModel()
- tuple **AMPERE** = units.BaseUnit("A")
  Unit instance to model the BaseUnit Ampere.

- tuple **BECQUEREL** = units.AlternateUnit("Bq", ∼SECOND)
  Unit instance to model the SI unit Becquerel.

- tuple **CANDELA** = units.BaseUnit("cd")
  Unit instance to model the BaseUnit Candela.

- float **CELSIUS** = 273.15
  Unit instance to model the SI unit degree Celsius.

- tuple **COULOMB** = units.AlternateUnit("C", AMPERE ∗ SECOND)
  Unit instance to model the SI unit Coulomb.

- tuple **FARAD** = units.AlternateUnit("F", COULOMB / VOLT)
  Unit instance to model the SI unit Farad.

- tuple **GRAY** = units.AlternateUnit("Gy", JOULE/KILOGRAM)
  Unit instance to model the SI unit Gray.

- tuple **HENRY** = units.AlternateUnit("H", WEBER / AMPERE)
  Unit instance to model the SI unit Henry.

- tuple **HERTZ** = units.AlternateUnit("Hz", ∼SECOND)
  Unit instance to model the SI unit Herz.

- tuple **JOULE** = units.AlternateUnit("J", NEWTON ∗ METER)
  Unit instance to model the SI unit Joule.
- tuple **KATAL** = units.AlternateUnit("kat", MOLE/SECOND)
  Unit instance to model the SI unit Katal.
- tuple **KELVIN** = units.BaseUnit("K")
  Unit instance to model the BaseUnit Kelvin.
- tuple **KILOGRAM** = units.BaseUnit("kg")
  Unit instance to model the BaseUnit Kilogram.
- tuple **LUMEN** = units.AlternateUnit("lm", CANDELA∗STERADIAN)
  Unit instance to model the SI unit Lumen.
- tuple **LUX** = units.AlternateUnit("lx", LUMEN/(METER∗METER))
  Unit instance to model the SI unit Lux.
- tuple **METER** = units.BaseUnit("m")
  Unit instance to model the BaseUnit Meter.
- tuple **MOLE** = units.BaseUnit("mol")
  Unit instance to model the BaseUnit Mol.
- tuple **NEWTON** = units.AlternateUnit("N", KILOGRAM ∗ METER/(SECOND ∗∗ 2))
  Unit instance to model the SI unit Newton.
- tuple **OHM**
- tuple **PASCAL** = units.AlternateUnit("Pa", NEWTON / (METER ∗∗ 2))
  Unit instance to model the SI unit Pascal.
- tuple **RADIAN** = units.AlternateUnit("rad", units.ONE)
- tuple **SECOND** = units.BaseUnit("s")
  Unit instance to model the BaseUnit Second.
- tuple **SIEMENS** = units.AlternateUnit("S", AMPERE / VOLT)
  Unit instance to model the SI unit Siemens.
- tuple **SIVERT** = units.AlternateUnit("Sv", JOULE/KILOGRAM)
  Unit instance to model the SI unit Sivert.
- tuple **STERADIAN** = units.AlternateUnit("sr", units.ONE)
- tuple **TESLA** = units.AlternateUnit("T", WEBER / (METER∗∗2))
  Unit instance to model the SI unit Tesla.

- tuple **VOLT** = units.AlternateUnit("V", WATT / AMPERE)
  Unit instance to model the SI unit Volt.
- tuple **WATT** = units.AlternateUnit("W", JOULE / SECOND)
  Unit instance to model the SI unit Watt.
- tuple **WEBER** = units.AlternateUnit("Wb", VOLT ∗ SECOND)
  Unit instance to model the SI unit Weber.

5.8 testcases.py File Reference

5.8.1 Detailed Description

This file contains a variety of test cases that verify this library.

Author:

```
Thomas Reidemeister
```

Namespaces

- namespace **scuq::testcases**

Classes

- class **TestArithmetic**
  This class provides the tests to verify the rational number module.
- class **TestComplexUncertaintyComponents**
  This class provides test-cases for the Module cucomponents.
- class **TestComplexUncertaintyComponents::OperationTest**
  This is the abstract super class for testing all operations of the Module cucomponents.
- class **TestGUMTree**
  These classes test the function of the global elements of the GUM-tree, namely the Context class.
- class **TestOperators**
  Test the unit conversion operators.
- class **TestQuantity**
  This class provides the test cases for the quantities.

- class **TestSIUnits**
  SI Testing class. This class tests the definition and semantics of the SI units.
- class **TestUncertaintyComponents**
  This class provides tests for the ucomponents module.
- class **TestUncertaintyComponents::Element**
  A class that is used for testing for identity.

Functions

- def **test_serialization**
  A general test for serialization of instances.

Variables

- tuple **suite** = unittest.TestSuite()

5.9 ucomponents.py File Reference

5.9.1 Detailed Description

This file contains the module to model uncertain values.

Author:

```
Thomas Reidemeister
```

Namespaces

- namespace **scuq::ucomponents**

Classes

- class **Abs**
  This class models the GUM-tree-nodes that take the absolute value of a silbling.
- class **Add**
  This class models GUM-tree nodes that add two silblings.
- class **ArcCos**
  This class models the GUM-tree-nodes that take the Arcus Cosine of a silbling.
- class **ArcCosh**

  This class models the GUM-tree-nodes that take the inverse Hyperbolic Cosine.
- class **ArcSin**
  This class models the GUM-tree-nodes that take the Arc Sine of a silbling.
- class **ArcSinh**
  This class models the GUM-tree-nodes that take the inverse Hyperbolic Sine of a silbling.
- class **ArcTan**
  This class models the GUM-tree-nodes that take the Arcus Tangent of a silbling.
- class **ArcTan2**
  This class models the inverse two-argument tangent.
- class **ArcTanh**
  This class models the GUM-tree-nodes that take the inverse Hyperbolic Tangent of a silbling.
- class **BinaryOperation**
  The abstract base class for modelling binary operations. This class provides the abstract interface for GUM-tree-nodes that have two silblings.
- class **Context**
  This class provides the context for an uncertainty evaluation. It maintains the correlation between the inputs and can be used to evaluate the combined standard uncertainty, as shown below. Let your model be $y = f(x_1, x_2, \ldots, x_N)$*, then* $u_c^2(y) = \sum_{i=1}^{N} \left(\frac{\delta f}{\delta x_i}\right)^2 u^2(x_i) + 2\sum_{i=1}^{N}\sum_{j=i+1}^{N} \frac{\delta f}{\delta x_i}\frac{\delta f}{\delta x_j} u(x_i, x_j)$.
- class **Cos**
  This class models the GUM-tree-nodes that take the Cosine of a silbling.
- class **Cosh**
  This class models the GUM-tree-nodes that take the Hyperbolic Cosine of a silbling.
- class **Div**
  This class models GUM-tree nodes that divide two silblings.
- class **Exp**
  This class models the GUM-tree-nodes that take the exponential of a silbling.
- class **Log**
  This class models the GUM-tree-nodes that take the Natural Logarithm of a silbling.
- class **Mul**
  This class models GUM-tree nodes that multiply two silblings.

- class **Neg**

  This class models the unary negation as GUM-tree-element.

- class **Pow**

  This class models GUM-tree nodes that raise the left silbling to the power of the right one.

- class **Sin**

  This class models the GUM-tree-nodes that take the Sine of a silbling.

- class **Sinh**

  This class models the GUM-tree-nodes that take the Hyperbolic Sine of a silbling.

- class **Sqrt**

  This class models the GUM-tree-nodes that take the square root of a silbling.

- class **Sub**

  This class models GUM-tree nodes that take the difference of the two silblings.

- class **Tan**

  This class models the GUM-tree-nodes that take the Tangent of a silbling.

- class **Tanh**

  This class models the GUM-tree-nodes that take the Hyperbolic Tangent of a silbling.

- class **UnaryOperation**

  The abstract base class for modelling unary operations. This class provides the abstract interface for GUM-tree-nodes that have one silbling.

- class **UncertainComponent**

  This is the abstract base class to model components of uncertainty as described in by "The GUM Tree".

- class **UncertainInput**

  This class provides the model for uncertain inputs, that are referred to as "Leafs" in "The GUM tree".

Functions

- def **clearDuplicates**

  Remove identical elements from a list.

5.10 units.py File Reference

5.10.1 Detailed Description

This file is evaluated whenever the units module is loaded.

It loads the classes necessary to operate the units module and performs some global initialization.

Author:

```
Thomas Reidemeister
```

Namespaces

- namespace **scuq::units**

Classes

- class **__ProductElement__**

  *A helper class for **ProductUnit** (p. 176) classes. This class helps to maintain the factors of a product unit.*

- class **AlternateUnit**

  This class provides an interface for units that describe the same dimension as another unit, but need to be distinguished from it by another symbol (e.g. to abbreviate them, or to distinguish their purpose).

- class **BaseUnit**

  This class provides the interface to define and use base units.

- class **CompoundUnit**

  This class provides an interface for describing compound units. The units forming a compound unit have to describe the same physical dimension. For example time $[hour : min : second]$.

- class **DerivedUnit**

  This class provides an abstract interface for all units that have been transformed from other units.

- class **Dimension**

  This class provides an interface to model physical dimensions.

- class **PhysicalModel**

  This class models the abstract interface for physical models.

- class **ProductUnit**

  The unit is a combined unit of the product of the powers of units.

- class **TransformedUnit**

  This class provides an interface for a unit that has been derived from a unit using an operator.

- class **Unit**

  An abstract class to model physical units.

- class **UnitsManager**

  This manages the alternate and base units as well as the physical dimensions.

Functions

- def **get_default_model**

  Get the physical model currently in use. This function returns None, if no model is currently in use.

- def **set_default_model**

  Set the default physical model to use.

Variables

- tuple **__char** = __unicode.encode("UTF-8")
- string **__unicode** = u"\u03b8"
- tuple **__UNITS_MANAGER__** = UnitsManager()

  Global units Manager that keeps track of the units and dimensions created.

- tuple **CURRENT** = Dimension("I")

  Predefined global dimension for the Electric Current.

- tuple **LENGTH** = Dimension("L")

  Predefined global dimension for the Length.

- tuple **LUMINOUS_INTENSITY** = Dimension("Li")

  Predefined global dimension for Luminous Intensity.

- tuple **MASS** = Dimension("M")

  Predefined global dimension for the Mass.

- tuple **NONE** = Dimension(ONE)

  Predefined global dimension for a dimensionless quantity.

- tuple **ONE** = ProductUnit()

  Dimensionless unit ONE.

- tuple **SUBSTANCE** = Dimension("n")

  Predefined global dimension for the Amount of Substance.

- tuple **TEMPERATURE** = Dimension(__char)

  Predefined global dimension for the Temperature.

- tuple **TIME** = Dimension("t")

  Predefined global dimension for the Time.

6 SCUQ Example Documentation

6.1 AlternateUnits.py

This example shows how to create and use instances of **units.AlternateUnit** (p. 56). They are created from other units using transformations. A symbol is assigned to the new unit, to distinguish it from other units. This symbol has to be unique and must not interfer with symbols of other units already created. We will show this by a simple example using SI units.

See also:

units.AlternateUnit (p. 56)

```
# You have to import this module to use si units.
from scuq import *

# Lets at first demonstrate what happens if a unit
# symbol is used twice.

try:
    myNewton = units.AlternateUnit("N",
                          si.KILOGRAM*si.METER/si.SECOND**2)
except qexceptions.UnitExistsException, exp:
    # "The following base unit has already been defined : N"
    # ... you should not create units having the same
                # symbol twice.
    print str(exp)

# Define the unit dynamic
myNewton = units.AlternateUnit("MN",
                           si.KILOGRAM*si.METER/si.SECOND**2)

# "MN"
print str(myNewton)

# "False", since they have different symbols
print "myNewton == si.NEWTON: "+str(myNewton == si.NEWTON)

# However they describe the same physical dimension, thus...
print "myNewton.is_compatible(si.NEWTON): "\
       +str(myNewton.is_compatible(si.NEWTON))
```

6.2 dft_example.py

In this section we show how to integrate SCUQ in NumPys fft module. Although we implemented most of NumPys ufuncs, we cannot use quantities directly in the fft module of NumPy. The reason for this drawback is the fft module being directly implemented in C requiring floating point ndarrays as input parameters. In compensation, we implemented the floating-point conversion functions of NumPy. Thus our type quantity can be converted to a floating-point number. In order to avoid an unwanted conversion, we require weak consistency checking be enabled to perform the conversion. In Line 5 we create an array of input data. These values are quantities and thus have a unit. To convert these values to float strict consistency checking has to be disabled as shown in Line 15. The converted array can be used as usual, however it lost the information about the unit. We suggest saving the default unit from the quantity before conversion takes place and reassign it to the result.

```
from numpy import *
from scuq import *

# generate some data, of a measured quantity
data = array([quantities.Quantity(si.VOLT, cos(50 * t)) \
              for t in xrange(1000)])

# the problem is, NumPy doesn't accept object
# arguments for the fft module; therefore,
# the data must be converted to the numpy
# float type.

# Enable weak consistency checking to convert to float64,
# the python floating point type.
quantities.set_strict(False)
f_data = float64(data)
quantities.set_strict(True)

# perform fft
ff_data = fft.fft(f_data)

# the result will also be the numpy complex type
assert(ff_data.dtype == complex)
```

6.3 ProductUnits.py

This example shows how to create and use instances of **units.ProductUnit** (p. 176). In general, instances of this class are not created directly. They are created by multiplying several other units. We will show this by a simple example using SI units.

See also:

units.Unit.__mul__ (p. 318)
units.ProductUnit (p. 176)

```
# You have to import this module to use si units.
from scuq import *

# You can create product units using other instances of
# units:
```



```
myUnit = si.KILOGRAM*si.METER/si.SECOND**2
# This creates a unit that has the same physical dimension
# as the unit Newton.

# kg*m*s^(-2)
print str(myUnit)

# Lets test if they are equal...
print "si.NEWTON == myUnit: "+str(si.NEWTON == myUnit)
# Unexpectedly this returns False, but why?
# si.Newton is an AlternateUnit, it could have been the case
# that myUnit has not the same purpose as si.NEWTON.

# However ...
print "si.NEWTON.is_compatible(myUnit): "\
       +str(si.NEWTON.is_compatible(myUnit))
# They may have different purposes, but they describe the same
# physical dimension. Therefore, they are compatible.

# That means that one could convert among them.
operator = si.NEWTON.get_operator_to(myUnit)
print "operator.convert(1) = "+str(operator.convert(1))
# ... In this case the operator returns the identical value
# (as expected).

# You can also do the above thing with an other unit than a
# base unit.
myUnit = si.NEWTON * si.METER

# N*m
print myUnit

# Lets test if they are equal...
print "myUnit == si.KILOGRAM*si.METER**2/si.SECOND**2: "+ \
       str(myUnit == si.KILOGRAM*si.METER**2/si.SECOND**2)
# Unexpectedly this returns False, but why?
# si.Newton is again an AlternateUnit.

# However ...
print "myUnit.isCompatible(si.KILOGRAM*si.METER**2/si.SECOND**2): "+ \
       str(myUnit.is_compatible(si.KILOGRAM*si.METER**2/si.SECOND**2))
# They may have different purposes, but they describe the same
# physical dimension. Therefore, they are compatible.

# As expected...
operator = myUnit.get_operator_to(si.KILOGRAM*si.METER**2/si.SECOND**2)
print "operator.convert(1) = "+str(operator.convert(1))

# This is to show that our library always maintains the Canonical
# form of a product of units.
myNewton = si.KILOGRAM*si.METER/si.SECOND**2

# yields: kg*m*s^(-2)
print str(myNewton)

# now lets create a unit that is compatible to si.PASCAL
myPascal = myNewton / si.METER**2

# yields: kg*m^(-1)*s(-2)
print str(myPascal)

# what happens if...
```

```
tmp = myPascal / myNewton
print str(tmp)
# should print: m^(-2)

# another trick:
tmp = myPascal / myPascal
print str(tmp)
assert(tmp == units.ONE)
#yields the neutral element
```

6.4 TransformedUnits.py

This example shows how to use the instances of **units.TransformedUnit** (p. 281). In general, it is not necessary to instance the **units.TransformedUnit** (p. 281) class directly. Instances of **units.TransformedUnit** (p. 281) are intended to be created implicitly by applying transformation to other units (i.e. BaseUnits). We will show this here by transforming a SI base unit.

See also:

units.Unit.__add__ (p. 316)
units.Unit.__div__ (p. 317)
units.Unit.__mul__ (p. 318)
units.TransformedUnit (p. 281)

```
# You have to import this module to use SI units.
from scuq import *

# Transformed units can be defined in the same physical dimension
# as a base unit by using the transformations below.

# 1. Adding an offset to a unit (i.e. degrees Celsius based on
#    Kelvin)
celsius        = si.KELVIN + 273.15
# 2. Dividing it by a constant value (i.e. dyn based on Newton)
dyn            = si.NEWTON / 100000
# 3. Muliplying it by an absolute constant value (i.e.
#    pound force from Newton)
lbf            = si.NEWTON * 4.4482216152605

# These transformations allow conversion among the
# units that describe the same physical dimension.

# This converts the unit back to the system unit, which is
# Kelvins in this case.
operator = celsius.to_system_unit()
result = operator.convert(1)
assert(result == -272.15)

# the same again for dyn -> Newton...
operator = dyn.to_system_unit()
result = operator.convert(1)
assert(result - 1e5 < 1e-10)

# the same again for lbf -> Newton...
operator = lbf.to_system_unit()
result = operator.convert(1)
assert(result - 0.2248089431 < 1e-10)
```



```
# You can also get an operator to convert among the
# transformed units: dyn -> lbf
operator = dyn.get_operator_to(lbf)
result   = operator.convert(1)
assert(result - 444822 < 0.5)
```

6.5 UncertainQuantity.py

This example shows how uncertain values can be used as quantities. Instead of encapsulating instances of **`quantities.Quantity`** (p. 184) inside an instance of **`ucomponents.UncertainInput`** (p. 309), you should always encapsulate uncertain values inside quantities. Otherwise this will lead to unpredicable behavior.

See also:

quantities (p. 29) The **quantities** (p. 29) module.
ucomponents (p. 33) The module to evaluate the uncertainty of scalar models.
cucomponents (p. 26) The module to evalute the uncertainty of complex-valued models

Author:

Thomas Reidemeister

```
# ATTENTION: You must NOT encapsulate quantities in uncertain
# components.
# this violates our design. Instead use the followin approach...

# You have to import this module to use quantities and uncertain
# values.
from scuq import *
# You have to import this module to use NumPy
import numpy as n

# You have to define the uncertain value first.
# This creates an uncertain value 1.0+-0.2
uvalue = ucomponents.UncertainInput(1.0, 0.2)

# Now you may encapsulate it in a quantity
# This creates a quantity (1.0+-0.2) [m]
uquantitiy = quantities.Quantity(si.METER, uvalue)

# Now you may build a model
# This creates a quantitiy that has m^(1/2) as unit and
# the propagation of the uncertainty is also performed.
model_1 = n.sqrt(uquantitiy)
assert(model_1.get_default_unit() == n.sqrt(si.METER))

# Create a context for the uncertainty evaluation.
c = ucomponents.Context()

# Evaluate the input VALUE
u_c = c.uncertainty(uvalue)
assert(u_c == 0.2)          # ...as expected

# Evaluate the input QUANTITY
u_c = c.uncertainty(uquantitiy)
```

```
# this one has a unit (!)
assert(u_c == quantities.Quantity(si.METER,0.2))

# Evaluate the model
u_c = c.uncertainty(model_1)
# ... the correct unit.
assert(u_c ==
  quantities.Quantity(n.sqrt(si.METER),0.1))
```

7 SCUQ Page Documentation

7.1 Coercion Rules

In this section we provide a complete set of coercion rules. These rules are used to convert among the data types of SCUQ to preserve the semantics. Coercion is performed whenever one of SCUQs types is involved in a binary operation. The goal of the coercion rules is to provide equal data types for both arguments of a binary operation; for example, the multiplication of a rational number and a floating point number should be performed by converting the rational number to a floating point number. Coercion is symmeric. Therefore the same applies to multiplications of floating point with rational numbers. We denote the rule as follows.

$$a \times f \rightarrow f(a) \times f$$

We denote the rules as follows:

- f and $f(x)$ refer to instances of `float`. The second argument is used to express the conversion of x to a `float`.
- z and $z(x)$ refer to instances of `long` and `int`. The second argument is used to express the conversion of x to a `long`.
- c and $c(x)$ refer to instances of `complex`. The second argument is used to express the conversion of x to a `complex`.
- nd refers to instances of `numpy.ndarray`.
- a and $a(x)$ refer to instances of **arithmetic.RationalNumber** (p. 210). The second argument is used to express the conversion of x to an instance of **arithmetic.RationalNumber** (p. 210). The conversion is implemented in **arithmetic.RationalNumber.value_of** (p. 231).
- q and $q(x)$ refer to instances of **quantities.Quantity** (p. 184). The second argument is used to express the conversion of x to an instance of **quantities.Quantity** (p. 184). The conversion is implemented in **quantities.Quantity.value_of** (p. 210).
- u_s and $u_s(x)$ refer to instances of **ucomponents.UncertainComponent** (p. 289). The second argument is used to express the conversion of x to an instance of **ucomponents.UncertainComponent** (p. 289). The conversion is implemented in **ucomponents.UncertainComponent.value_of** (p. 309).

- u_c and $u_c(x)$ refer to instances of **cucomponents.CUncertainComponent** (p. 117). The second argument is used to express the conversion of x to an instance of **cucomponents.CUncertainComponent** (p. 117). The conversion is implemented in **cucomponents.CUncertainComponent.value_of** (p. 133).
- u denotes an instance of **units.Unit** (p. 314).
- $\emptyset$ denotes an undefined operation (i.e. the coercion raises an exception).

The cohercion rules by type:

- Type: **arithmetic.RationalNumber** (p. 210)

$$\begin{aligned}
a \times a &\rightarrow a \times a && (1)\\
a \times z &\rightarrow a \times a(z) && (2)\\
a \times c &\rightarrow c(a) \times c && (3)\\
a \times f &\rightarrow f(a) \times f && (4)\\
a \times q &\rightarrow q(a) \times q && (5)\\
a \times u_s &\rightarrow u_s(a) \times u_s && (6)\\
a \times u_c &\rightarrow u_c(a) \times u_c && (7)\\
a \times u &\rightarrow \emptyset && (8)\\
a \times nd &\rightarrow \emptyset && (9)
\end{aligned}$$

- Type: **quantities.Quantity** (p. 184)

$$\begin{aligned}
q \times q &\rightarrow q \times q && (10)\\
q \times z &\rightarrow q \times q(z) && (11)\\
q \times c &\rightarrow q \times q(c) && (12)\\
q \times f &\rightarrow q \times q(f) && (13)\\
q \times u_s &\rightarrow q \times q(u_s) && (14)\\
q \times u_c &\rightarrow q \times q(u_c) && (15)\\
q \times nd &\rightarrow q \times q(nd) && (16)\\
q \times u &\rightarrow \emptyset && (17)
\end{aligned}$$

- Type: **ucomponents.UncertainComponent** (p. 289)

$$\begin{aligned}
u_s \times u_s &\rightarrow u_s \times u_s && (18)\\
u_s \times z &\rightarrow u_s \times u_s(z) && (19)\\
u_s \times f &\rightarrow u_s \times u_s(f) && (20)\\
u_s \times nd &\rightarrow \emptyset && (21)\\
u_s \times u_c &\rightarrow \emptyset && (22)\\
u_s \times c &\rightarrow \emptyset && (23)\\
u_s \times u &\rightarrow \emptyset && (24)
\end{aligned}$$

- Type: **cucomponents.CUncertainComponent** (p. 117)

$$u_c \times u_c \rightarrow u_c \times u_c \quad (25)$$
$$u_c \times z \rightarrow u_c \times u_c(z) \quad (26)$$
$$u_c \times f \rightarrow u_c \times u_c(f) \quad (27)$$
$$u_c \times c \rightarrow u_c \times u_c(c) \quad (28)$$
$$u_c \times nd \rightarrow \emptyset \quad (29)$$
$$u_c \times u \rightarrow \emptyset \quad (30)$$

Attention:

The binary operators from `numpy`, such as numpy.arctan2, and numpy.hypot, do not implement coercion. Instead, they broadcast arctan2 to the first argument. Therefore our coercion rules are not symmetric when using operators from numpy.

7.2 Files and Directories

In this section we describe the files that are not included in the file documentation.

AUTHORS This file contains the contact information of the authors of SCUQ.

At the time this document was created, only Thomas Reidemeister is involved.

doc.cfg Doxygen (see link below) uses this file to create the SCUQ reference manual automatically.

It contains the style- and output format definitions. Please do not directly invoke Doxygen on this file; use the respective make target instead (`make clean`).

Makefile This file is used by GNU make to assist the installation of SCUQ and perform a variety of administrative tasks.

- Performing the self test (`make test`).
- Creating the reference manual (`make doc`).
- Building backups of the current state of the library (`make backup`).
- Cleaning temporary files (`make clean`).

A companion file is `make_latex.sh`. It is a script to create the PDF documentation whenever `make doc` is invoked.

examples This directory contains the application examples described in this programming manual and in the Reidemeister thesis.

doc This directory is a placeholder for the reference manual if it is created from the source code. By default the documentation is created in PDF- and HTML format.

See also:

- SCUQ Installation Instructions
- SCUQ Example Documentation (included in this manual)
- "SCUQ - A Class Library for the Evaluation of Scalar- and Complex-valued Uncertain Quantities"; Thomas Reidemeister; Diploma-Thesis; Otto-von-Guericke University, Magdeburg, Germany (2007)
- GNU Make (`http://www.gnu.org/software/make/`)
- Doxygen (`http://www.stack.nl/~dimitri/doxygen/`)

7.3 Installation

In this section we describe the installation of SCUQ in the user space. Note that the version numbers of the tools refers to the minimum version required. SCUQ may also run using later versions.

Minimum Requirements:

- Python 2.4, installed and registered in the `PATH` environment variable.
- NumPy 1.0.1, installed as module in the Python distribution.
- A tool uncompressing zip files (e.g. Info-ZIP or 7-ZIP). We assume that a console application exists to unzip files from the command line that is registered in the `PATH` environment variable.

Optional Requirements:

- GNU Make, installed and registered in the `PATH` environment variable.
- GNU Tar, installed and registered in the `PATH` environment variable.
- Bzip2, installed and registered in the `PATH` environment variable.
- Doxygen 1.5.1, installed and registered in the `PATH` environment variable.
- Ghostscript 8.15.0, installed and registered in the `PATH` environment variable.
- BSD Shell, installed and registered in the `PATH` environment variable.
- LaTeX and PDFLaTex, installed and registered in the `PATH` environment variable.

Most of the optional tools required are included in recent Linux distributions and Cygwin. Doxygen can be obtained using the link shown below.

We describe two types of the installation:

- A minimal installation that installs SCUQ for the use in Python only.
- A comprehensive installation that installs the class library for the use in Python, performs the self-test, and generates the SCUQ reference manual. This installation can also generate backups of the current state of SCUQ (i.e. if it is modified by the user).

Minimal Installation

1. Copy the archive SCUQ.zip to the directory desired. We denote it as <your project dir>.
2. Open a console (e.g. BASH on Linux, CMD.EXE on Windows)
3. Change to the project directory using
   `cd <your project dir>`
4. Unzip the archive SCUQ.zip using
   `unzip SCUQ.zip`
   You may also use other tools to uncompress the archive.
5. The classes and modules are now unzipped into the directory
   `<your project dir>/SCUQ/`
6. Change to this directory using
   `cd SCUQ`
7. Verify the compatibility of your platform running the suite of test cases using
   `python` **`scuq/testcases.py`** (p. 342)
   The console output must not contain any exceptions. If it contains any exceptions then SCUQ will most likely not run on your system. These failures maybe due to a wrong configured Python installation or your platform does not meet the required floating-point accuracy.
8. If SCUQ passed the self-verification, you can use it in your software. Please copy the subdirectory `scuq` to the root of your project directory. Then you can import SCUQ using
   `from scuq import *`
   in your projects code.

Comprehensive Installation

1. Perform the steps 1-6 of the minimal installation
2. Create the documentation and perform the self-verification using
   `make`
   The output of this command must not print any errors. Errors maybe due to a wrong installation of the tools required or your platform does meet the required floating-point accuracy.

3. If SCUQ passed the self-verification, you can use it in your software. Please copy the subdirectory `scuq` to the root of your project directory. Then you can import SCUQ in your project using
   `from scuq import *`
   The programming manual in HTML and PDF format is stored in the subdirectory doc.

See also:

- Doxygen (`http://www.stack.nl/~dimitri/doxygen/`)
- Python (`http://www.python.org/`)
- GNU (`http://www.gnu.org/`)
- teTeX (`http://www.tug.org/teTeX/`)
- MiKTeX (`http://www.miktex.org/`)
- NumPy (`http://www.scipy.org/`)
- Ghostscript (`http://www.cs.wisc.edu/~ghost/`)
- Info-ZIP (`http://www.info-zip.org/`)
- 7-ZIP (`http://www.7-zip.org/`)
- CygWin (`http://cygwin.com/`)