RichValues
– Python Library –
User Guide

Andrés Megías Toledano
Centro de Astrobiología (CAB), Madrid

Version 4.2
August 2025

Index

1. Introduction

RichValues is a Python 3 library whose purpose is to manage numeric values with uncertainties, upper/lower limits and finite numeric intervals, which we will call *rich values*.[1] With this library, one can import rich values written in plain text documents in an easily readable format, operate with them propagating the uncertainties automatically, and export them in the same formatting style as the import. It also allows to easily plot rich values and to make fits to any function taking into account the uncertainties and the upper/lower limits or finite intervals. Moreover, correlations between variables (that is, variables that are not independent) are taken into account when performing operations between rich values.

For example, to represent the value $(6.3 \pm 0.4) \cdot 10^3$, we could write `6.3 +/- 0.4 e3`, and for the upper limit $< 1.2 \cdot 10^{-5}$ we would write `< 1.2 e-5`. Asymmetric uncertainties are also supported; for example, $4.2^{+0.5}_{-0.4}$ would be written as `4.2 -0.4+0.5`. We can even specify the domain of the variable between brackets, typing `inf` to represent infinity; for example, `3.8 +/- 0.5 e4 [0, inf]`. These text strings will be parsed as Python objects containing all the necessary information to describe the rich value, and they will be displayed in the screen in the same formatting style. The main way to do so is to write the text string between simple quotation marks and put it inside the function `rich_value` (which can also be called as `rval`); for example, `rich_value('4.2 +/- 0.4')` or `rval('0 +/- 0.3')`. Alternatively, one can directly input a probability density function (PDF) that represents the rich value, and the central value and the uncertainties will be estimated automatically. Integer rich values, corresponding to integer variables, and complex rich values, with complex numbers, are also supported.

Then, we could make arithmetic operations between them and other rich values or even usual numbers, like addition, subtraction, multiplication and division. For simple operations we can use the arithmetic operators (`+`, `-`, `*`, `/`, `**`, `//`, `%`), while for more complex operations we can specify the function to be applied in a simple way (see section 4.1.3). Rich values are characterized by probability distributions, allowing the correct calculation of the uncertainty propagation even for high uncertainties (see section 8 for more details, and also section 9.3). Also, correlations between variables that are not independent are taken into account for the calculations. Lastly, rich values can be used within arrays and tables (sections 4.2 and 4.3), which, combined with the plotting and fitting features (sections 6 and 7), makes this library suitable for scientific purposes.

2. Installation

RichValues works in Python 3. It requires the libraries NumPy, Pandas, SciPy and Matplotlib. This library is published on GitHub:

https://github.com/andresmegias/richvalues/ .

In order to install it, you can use the Python Package Installer (PyPI), running from the terminal the following command:

1. Like *rich text* for text with information about the font type, size, weight, etc.

```
pip3 install richvalues
```

You can also use the Conda package installer, running instead the following command:

```
conda install richvalues -c richvalues
```

Alternatively, you can install the library manually in your computer. To do so, download the `__init__.py` file from the `richvalues` folder of the GitHub repository, rename it to `richvalues.py` and copy it to one of the Python system paths (that file contains the source code of the whole library). To display the list of all these paths, you can run Python (for example, from the terminal, writing `python3`), and run the commands `import sys` and `sys.path` consecutively. For example, for MacOS, if you did not install Conda one of these paths would be:

```
/Users/<user>/Library/Python/<3.x>/lib/python/site-packages
```

with `<user>` being your username and `<3.x>` your exact Python version.

Now, RichValues can be used in a Python script like any other library, with the name `richvalues`. Therefore, to import it you would just write `import richvalues`. You can also use an abbreviation for the library name, like `rv`:

```
import richvalues as rv
```

In the following examples of code, we will consider that we already imported RichValues with this abbreviation. We will also use the abbreviations `np` and `pd` for the libraries NumPy and Pandas, respectively.

3. Formatting style for representing rich values

The RichValues library uses a specific syntax to represent the different kinds of rich values with plain text. This is the way in which they are displayed in the screen and in which they can be imported and exported. It can also be used to create rich values within a script. Below are the rules for this formatting style:

- If the value has no uncertainty, just put the number.
- It it has uncertainty, you can join the central value and the uncertainty with `+/-` or `+-`, using blank spaces if you want; for example: `5.2 +/- 0.4`.
- If it has a lower and an upper uncertainty, you should write the lower uncertainty just after the central value preceded by `-`, and then write the upper uncertainty preceded by `+`, using blank spaces if you want; for example: `5.2 -0.3+0.4`.
- If you want to use scientific notation (exponential notation with decimal base), for the central value and the uncertainty (or uncertainties), you just have to write an `e`, separated by a blank space from the numbers and followed by the exponent argument; for example: `5.2 -0.3+0.4 e-3`.
- If the uncertainty is very low, you can put the central value and then the uncertainty between brackets, and its level of decimals will be assumed to be the same as the central value; for example: `1.000436(21)`, `42103(8)`.
- If you want to specify an upper or lower limit, just write `<` or `>` before the value (without uncertainty), putting a blank space if you want; for example: `< 5.2 e2`.

- If you want to specify the domain of the value, that is, the minimum and maximum values that the magnitude corresponding to this value could take, you have to write it between brackets, using the text `inf` to represent infinity; for example: `5.2 -0.3+0.4 e3 [0,inf]`. By default (that is, if it is not specified), the domain is all the real numbers, that is: `[-inf,inf]`. You can also use `pi` for π and `tau` for τ (2π), and can write simple expressions like `[-tau/2,tau/2]`.
- If you wanted to specify a finite interval of values, you should write the edges of the interval separated by two hyphens (--), without uncertainties; for example: `9 e3 -- 60 e3`.

4. Creation of rich values

With the library RichValues, we can create individual rich values and use them in tuples, lists, dictionaries, etc., but we can also create arrays –based on NumPy arrays– and tables –based on Pandas dataframes.

4.1. Individual rich values

There are mainly two ways to create a single rich value: with the function `rich_value` and with the class `RichValue`, being the first one the easiest way.

4.1.1. Function `rich_value`

It can also be called with the shortened name `rval`. Below are the arguments of this function, although the first one is the only required:

- **`text`**. Text string representing the rich value, using the formatting style explained in section 3. It can also be a number, but then it will be treated as an exact rich value, with no uncertainty.
- **`domain`**. List containing the edges of the domain of the rich value, that is, the minimum and maximum values that the magnitude corresponding to this array could take. By default, it is the domain written in the input text string, but if it is not written it will be (–∞, ∞).[2]
- **`is_int`**. Logical variable used to indicate that the magnitude corresponding to the rich value is integer, even if the main value or the uncertainties are not integer. The default is `False`.
- **`pdf`**. Optional variable containing the probability density function (PDF) that represents the rich value. This can be used instead of writing the rich value in the formatting style of the previous section, as their properties (main value, uncertainties, etc) will be automatically derived from the PDF. It can be given as a Python function, as a dictionary containing the values of the variable (with key `values`) and the corresponding values of the PDF (with key `probs`), or a list, tuple or array with the same two sets of values, with the variable values first and then the PDF values.
- **`consider_intervs`**. This logical variable is only needed if a PDF is written to represent

2. The NumPy object `inf` is used to represent infinity (∞).

the rich value. It indicates if the rich value can be an upper/lower limit or a finite interval, or it can only be a number with uncertainties.

- **use_default_extra_sf_lim**. This is an optional variable related to the number of significant figures to show when displaying the rich value, and is only important when using the function hundreds or thousands of times, as it speeds up a little bit the reading of the rich values. This determines which *limit for extra significant figure* is used for displaying the rich value (see section 10 for more details). If `use_default_extra_sf_lim` is `True`, the default value for the limit for extra significant figure (`2.5`) will be used, but if it is `False`, this value will be inferred by the input text that represents the rich value.

The default values for these arguments, and for most of the arguments of the following functions, can be modified as explained in section 7.

Below is a simple example of how we would create two different rich values, using the shortened name of the function:

```
x = rv.rval('5.2 +/- 0.3')
y = rv.rval('2.1 -0.4+0.5')
```

In this way, we have created a Python object of class `RichValue`. These objects will be displayed on screen with the previous formatting style except for the domain, which will be hidden. For example:

```
rv.rval('0.327 -0.12+0.18 [-1,1]')
[out]   0.33-0.12+0.18
```

Note that when showing the rich value the central value has been rounded to display the same number of decimals than the uncertainties. However, the rich value stores the full number, which would be used for any calculations. That is, rich values always store the numbers that define it with all its decimals, the rounding is only done for displaying purposes and it is not really affecting the properties of the rich value.

4.1.2. Class `RichValue`

This is the main class of the library. The other classes and most of the functions are built around it.

Arguments

Below are the arguments of this Python class, being the first one the only required one:

- **main**. Main value of the rich value, that is, central value or value of the upper/lower limit.
- **unc**. Uncertainty associated with the central value. If two values are given, they would be the lower and upper uncertainties, respectively. By default it is `0`.
- **domain**. List containing the edges of the domain of all the entries of the rich array, that is, the minimum and maximum values that the magnitude corresponding to this array could take. By default, it is `[-np.inf, np.inf]`.[3]

3. Assuming that the library NumPy was imported with the abbreviation `np` (`import numpy as np`). Otherwise, one can also use `rv.inf`, which is a shortcut to `np.inf`.

- **is_lolim**. Logical variable that determines if the rich value is a lower limit. By default it is `False`.
- **is_uplim**. Logical variable that determines if the rich value is an upper limit. By default it is `False`.
- **is_range**. Logical variable that determines if the rich value is actually a constant range of values.[4] By default it is `False`.
- **is_int**. Logical variable used to indicate that the magnitude corresponding to the rich value is integer, even if the main value or the uncertainties are not integer. The default is `False`.

Most of the argument names have expanded names that can be used instead of the ones shown before (see section 9). For example, instead of `unc` you can write `uncertainty`.

As an example, below is a code to create the same rich values as in the examples of the previous section:

```
x = rv.RichValue(5.2, 0.3)
y = rv.RichValue(2.1, [0.4,0.5], domain=[0,np.inf])
z = rv.RichValue(0.32, [0.12,0.18], domain=[-1,1])
```

Instance variables

All of the arguments of the `RichValue` class correspond to instance variables with the same name.[5] To access to them, you should write a dot after the name of the rich value Python variable and then the name of the instance variable. For example, to access to the instance variable `main` of a certain rich value `x`, you would write `x.main`. You can modify these variables if you want to modify the rich value. Additionally, there are other four instance variables for this class:

- **num_sf**. Number of significant figures to use for displaying the elements of the rich array, taking into account the uncertainties. By default it is `1`.
- **min_exp**. Minimum exponent in absolute value to be used for displaying the rich value in scientific notation. By default it is `4`.
- **max_dec**. Maximum number of decimals to be displayed, to use the notation with parenthesis instead. By default it is `5`.
- **extra_sf_lim**. Value of the *limit for extra significant figure* that will be used for displaying the rich value in scientific notation. If the significand/mantissa[6] of the uncertainty of the rich value (if it is a central value with uncertainty) or its main value (if it is an upper/lower limit, or an edge of an interval) is lower than this limit number, the rich value will be displayed with an additional significant figure. By default, it is `2.5`.
- **pdf_info**. Information about the probability density function (PDF) representing the rich value. If the rich value was created specifying a PDF instead of a text, that func-

4. This can be specified instead with the argument `mains`, putting the edges of the interval as a list.
5. An instance variable is a variable contained by an object of a certain class.
6. The number multiplying the decimal power.

tion will be stored here. If not, this value will be the text `default`, which means that the default PDF function will be used (see section 8 for more details).

- **`variables`**. List containing the information of the variables on which the current rich value depends, with arbitrary names. If it is an independent variable (a new rich value created from scratch), it will contain just a name that identifies this rich value (an `x` followed by an arbitrary number). However, if this variable comes from applying an operation to other rich values, the names of those variables will also be stored here.
- **`expression`**. Text string that contains the mathematical formula that describes the relations between the variables on which the current rich value depends, using the variable names of the `vars` instance variable.

The first three instance variables determine how the rich value is displayed on screen, and also how it will be exported into a plain text file. The other two instance variables are used for handling the correlation between variables when performing operations between rich values.

As for the argument names for the `RichValue` class, you can use alternative longer names instead (see section 9). For example, instead of `num_sf` you can write `number_of_significant_figures`.

Attributes

Apart from the instance variables, the `RichValue` class has several attributes.[7] They correspond to properties of the rich value that describe it. The way to access them is the same as with instance variable, that is, writing a dot followed by the attribute name. Below is a list with all the attributes of this class:

- **`is_lim`**. Logical variable that is `True` if the rich value is an upper/lower limit.
- **`is_interv`**. Logical variable that is `True` if the rich value is a finite interval of values or an upper/lower limit. It is the opposite as the result of `is_centr(self)`.
- **`is_centr`**. Logical variable that is `True` if the rich value is a centered value, with a central value and uncertainties. It is the opposite as the result of `is_interv(self)`.
- **`is_exact`**. Logical variable that is `True` if the rich value is a centered value and it has no uncertainty, that is, an exact value.
- **`is_const`**. Logical variable that is `True` if the rich value is an exact value and its domain is just the main value, that is, a constant value which cannot have a different value; for example, a mathematical constant like π or τ.
- **`is_finite`**. Logical variable that tells if the rich value is corresponds to a centered value, an upper/lower limit or a constant range of values, but not $\pm\infty$ nor NaN.
- **`is_inf`**. Logical variable that tells if the rich value is infinity (∞) or minus infinity ($-\infty$).
- **`is_nan`**. Logical variable that tells if the rich value is a NaN (*not a number*).
- **`center`**. If the rich value is a centered value, it returns the main value (that is, the cen-

7. An attribute is a variable related to an object of a certain class that is calculated on the fly when it is called.

tral value); if not, it returns a NaN (*not a number*; in particular: `np.nan`).[4] This can be useful when using Matplotlib's function `errorbar`.[8]

- **`unc_eb`**. Uncertainties of the rich value as a list with shape (2, 1), so that it can be easily used with Matplotlib's function `errorbar`.[8]
- **`rel_unc`**. Relative uncertainties of the rich value.
- **`signal_noise`**. Signal-to-noise ratios (S/N) of the rich value, which are the inverses of the relative uncertainties.
- **`ampl`**. Distances between the central value and the bounds of the domain, which we call *amplitudes*.
- **`rel_ampl`**. Distances between the central value and the bounds of the domain divided by the uncertainties (*relative amplitudes*). They are a measure of the normality of the distribution associated with the rich value (the greater these values are, the bigger the similarity to a normal distribution will be).
- **`norm_unc`**. Uncertainties divided by the distances between the central value and the bounds of the domain (*normalized uncertainties*). They are a measure of the normality of the PDF associated with the rich value (the less these value are, the bigger the similarity to a normal distribution will be).
- **`prop_score`**. It returns the *propagation score*, defined as the minimum value of the signal-to-noise ratios and the relative amplitudes of the rich value. This value will be used to determine the use or not of a fast approximation to calculate the uncertainty propagation when applying a function to the rich value, in case that the propagation score is high enough.

As for the argument and instance variable names for the `RichValue` class, you can use alternative longer names instead (see section 9). For example, instead of `rel_unc` you can write `relative_uncertainty`.

Methods

The `RichValue` class has several methods.[9] Some of them just provide additional information on the rich value but others are more complex and can be used to modify the rich value. Below is the list of all of them with their arguments between brackets:

- **`interval`**(self, sigmas).[10] If the rich value is a central value with uncertainties, it returns the interval defined by the central values plus and minus `sigma` times the corresponding uncertainties (by default, `sigmas = 3.0`). If the rich value is a constant interval of values or an upper/lower limit, it returns the interval of values that defines the rich value (which can include infinity as one of the edges).
- **`sign`**(self, sigmas). It returns the sign associated with the rich value. To do so, the in-

8. You can instead use the function `errorbar` within this library (see section 6).
9. A method is a function that can be called within an object of a certain class.
10. The arguments self means that the method has to be called within an object of this class, but it is not actually written when calling the method; for example: if `x` is a rich value, to call the method `rel_unc` we should write `x.rel_unc()`.

terval method is used, using the given sigmas as its argument (by default, sigmas = np.inf). It first calculates the interval of possible values of the rich value (with the interval method). Then, if they are all positive or zero, the result will be 1; if they are all negative or zero, it will be –1; if they are all zero (the rich value is exactly 0), it will be 0; in any other case, the result will be undefined (np.nan).

- **median**(self, num_points). Median of the PDF of the rich value. If it is a centered rich value and the default PDFs are used, this should be equal to the main value of the rich value. If the argument num_points is specified, it will be estimated numerically creating a sample of that size.
- **mean**(self, num_points, sigmas). Mean of the PDF of the rich value. The parameter sigmas defines the interval that is used to compute the mean, while num_points defines its spatial resolution. The default values are num_points = int(1e4) and sigmas = 8.0.
- **mode**(self, num_points, sigmas). Mode of the PDF of the rich value. The two optional parameters are the same as in the method mean.
- **var**(self, num_points, sigmas). Variance of the PDF of the rich value. The two optional parameters are the same as in the method mean.
- **std**(self, num_points, sigmas). Standard deviation of the PDF of the rich value. The two optional parameters are the same as in the method mean.
- **skewness**(self, num_points, sigmas). Skewness of the PDF of the rich value. The two optional parameters are the same as in the method mean.
- **kurtosis**(self, num_points, sigmas). Kurtosis of the PDF of the rich value. The two optional parameters are the same as in the method mean.
- **excesss_kurtosis**(self, num_points, sigmas). Kurtosis of the PDF of the rich value, defined as the kurtosis minus 3.
- **moment**(self, n, central, standarized, num_points, sigmas). Moment of order n (n-moment) the PDF of the rich value. If central = True, it will be a central moment, that is, with respect to the mean. If standarized = True, it will be standarized, that is, divided by the standard deviation to the power of n. The two last optional parameters are the same as in the method mean.
- **latex**(self, show_dollars, mult_symbol, use_extra_sf_in_exacts, omit_ones_in_sci_notation). It returns a text string of a LaTeX code representing the rich value, using mult_symbol as the multiplication symbol in scientific notation (by default, mult_symbol = \\cdot).[11] The logical variable show_dollars, which is True by default, determines if the text string is enclosed with dollar symbols ($) or not. The other two arguments, that are optional, refer to the default options use extra significant figure for exact values and omit ones in scientific notation in LaTeX (see section 10).
- **set_lim_unc**(self, factor). If the rich value is an upper/lower limit, the lower/upper uncertainty (stored in the instance variable unc) will be replaced with the central value

11. In Python, two backslashes are needed in order to display just one in a text string.

divided by factor (by default, factor = 4.0). This can be useful when plotting upper/lower limits with the function errorbar from the library Matplotlib.[8]

- **pdf**(self, x). It applies the probability density function (PDF) associated with the rich value to the given array x, returning the resulting array.
- **sample**(self, N). It returns a sample of size N of the distribution associated with the rich value, whose corresponding PDF can be explored with the previous method, pdf. By default, N is equal to the 8 000. If the rich value is of integer nature, the values of the sample will be integers.
- **function**(self, function, **kwargs). It applies the given function (function) to the rich value. Additional arguments can be passed (kwargs), which are the arguments of the function function with_rich_values (see next section).

The last methods allows to apply a function to the rich value and therefore to obtain a new one. Next section explains how to make operations with rich values.

4.1.3. Operations with rich values

The RichValue class has special methods for the basic arithmetic operations: addition, subtraction, multiplication, division, power, integer division and module. Therefore, you can just use the arithmetic operators (+, -, *, /, **, //, %) to combine a rich value with another rich value or with a usual number. The uncertainties will be propagated automatically, and the condition of an upper/lower limit (or even a finite interval) will be taken into account to obtain the final result. For example:

```
x = rv.rval('5.2 +/- 0.3')
y = rv.rval('2.1 -0.4+0.5')
x + y
[out]   7.3-0.5+0.6
x = rv.rval('< 5.2 [0,inf]')
x + 3
[out]   3.0 -- 8.2
x = rv.rval('5.2 +/- 0.3')
x**2
[out]   27+/-3
```

Additionally, correlations between variables are taken into account automatically. For example, if we compute a variable y = x + x, and then we compute y - 2*x, the result should be exactly zero. Similarly, the operation y + x should yield the same result as 3*x.

```
x = rv.rval('2.1 +/- 0.3')
y = x + x
print(y - 2*x)
[out]   0
print(y + x, 3*x)
[out]   6.3+/-0.9  6.3+/-0.9
```

Although you can concatenate several arithmetic operations and the correlation between vari-

ables will be preserved, if you want to apply more than two or three operations, it will be faster to use function_with_rich_values. Plus, it allows to apply any function to the input rich values. Besides of that, you can also check section 9.5 to see implemented mathematical functions from NumPy, like np.exp, np.log or np.sin.

Function function_with_rich_values

Using this function one can compute any expression involving rich values. It can also be called with the shortened name function. Below is the full list of its arguments, although only the two first ones are mandatory:

- **function**. Text string representing the function to be applied to the given rich values. It should be the source code of the expression of the function, using empty brackets to indicate the position of the arguments, in the same order as they are in the variable args. For example, '{} + {}' or 'np.sin({})'. Alternatively, you can just put a Python function, but then the correlation between the arguments will not be taken into account (the result of the function will be treated as a new independent variable). This can be fine if you know that the arguments of the function are actually independent. In this case, if its expression is short, it can be defined within the list of arguments using a lambda function.[12] In both cases (either using a text string or directly a Python function), the output of the given Python function can be a single value or several ones.
- **args**. List with the input rich values, in the same order as the arguments of the given function.
- **unc_function**. Python function used to approximate the uncertainties in case that analytic uncertainty propagation can be applied. This will only be used if function is a Python object instead of a string representing the function, that is, it is only useful if all the arguments (args) are independent variables. The arguments of this function should be the central values first and then the uncertainties, with the same order as in the input function.[13] If it is not specified, the analytic uncertainty propagation will never be used.
- **is_vectorizable**. Logical variable that states if the given Python function is vectorizable, that is, if it can be applied to NumPy arrays. If so, computation time will decrease considerably. By default it is False.
- **len_samples**. Size of the samples of the arguments that will be drawn for calculating the final distribution applying the input function. The default is the square root of the number of arguments times the default sample size (8 000).
- **domain**. Domain of the result. If the given function returns several outputs, it can be a list of the domains of the different outputs. If this variable is not specified, the domain of the output (or domains) will be estimated automatically.

12. A lambda function is defined in the following way: first you write lambda followed by a blank space; now, you write the arguments of the function with any desired letter and separated by commas, ending with a colon, and followed optionally by a blank space; finally, you write the mathematical expression of the function using the same letters you used before. For example: lambda a,b: a+b .

13. For example: lambda a,b,da,db: da+db .

- **is_domain_cyclic**. Logical variable that indicates if the domain of the result is finite and cyclic, that is, that both edges are equivalent. This happens for some trigonometric functions, like the arcsine, arccosine, etc. This argument needs to be `True` to properly propagate the values of the sample when they cross the domain edges; by default it is `False`.
- **sigmas**. Threshold to use approximate uncertainty propagation. The value is the minimum of the signal-to-noise ratios and the distances to the bounds of the domain relative to the uncertainties. By default it is `20.0`.
- **consider_intervs**. Logical variable that determines if the final distribution can be interpreted as an upper/lower limit or a constant range of values; if not, it will be treated as a rich value with a central value and uncertainties. By default it is `False` if all of the arguments are centered values and `True` if not.
- **use_sigma_combs**. Logical variable that determines if the calculation of the uncertainties is optimized when approximate uncertainty propagation can be performed but there is no uncertainty function provided. It performs combinations of the central value plus and minus the uncertainties for every argument and applies the function, taking the minimum and maximum of the resulting values as bounds to compute the uncertainties. By default it is `False`, as it is not fully tested for more than one argument.
- **force_approx_propagation**. If this logical variable is set to `True`, approximate uncertainty propagation (either with the provided analytic function or with combinations) will be used even if the propagation score is not high enough. By default it is `False`.
- **lims_fraction**. In case the resulting value is an upper/lower limit, this factor is used to calculate the limit. It can take values from 0 to 1, and the closest it is to 1, the closest the resulting limit will be to the function applied to the central/limit value of the arguments. By default it is `0.1`.
- **num_reps_lims**. Number of repetitions of the sampling done in the cases of having an upper/lower limit for better estimating its value. By default it is `4`. Greater values are recommended if `lims_fraction` is greater than `0.1`.
- **save_pdf**. This optional logical variable can be set to `True` to store the information of the shape of the resulting distribution (that is, the probability density function, PDF) into the resulting rich value. It will be saved as two sets of values inside a Python dictionary, one containing values of the variable (`values`) and the other with the corresponding PDF values (`probs`). This can be used to not loose information then performing operations with rich values, specially if the input rich values of the function have custom PDFs. By default this variable is `False`.

Let's see a short example of the use of this function, using its shortened name.

```
x = rv.rval('5.2 +/- 0.3')
y = rv.rval('2.1 -0.4+0.5')
z = rv.function('{}+{}', [x,y])
print(z)
[out]   7.3-0.5+0.6
```

That would be it. As you can see, the basic use of this function is quite simple. Now, if we subtracted the variable x from z, the result would be exactly the variable y, which can be checked inspecting the instance variable expression from the result.

```
zx = z - x
print(zx, zx.expression)
[out]   2.1 -0.4+0.5  ((x1)+(x2))+(-(x1))
```

The variable x1 in the output corresponds to x, and x2 corresponds to y, so you can see that the final expression is equivalent to just x2 (y). Now, if the given function has to return several outputs, the code would be very similar, we should just add brackets or parenthesis to indicate that there is more than one output from the function.

```
z2 = rv.function('({}+{}, {}-{})', [x,y,x,y])
print(z2)
[out]   (7.3-0.5+0.6, 3.1-0.6+0.5)
```

Lastly, if we wanted to directly, supply Python functions instead of text strings, the code would be also quite similar.

```
z  = rv.function(lambda a,b: a+b, [x,y])
z2 = rv.function(lambda a,b: [a+b,a-b], [x,y])
```

However, remember that in these cases the correlation between the function arguments is not preserved. Therefore, if we compute z - x, we would not obtain exactly y, with the main difference being in the uncertainty.

```
zx = z - x
print(zx, zx.expression)
[out]   2.1 +/-0.6  (x4)+(-(x1))
```

Indeed, you can see that now in the mathematical expression our variable z is considered as a new independent variable (called x4). Therefore, using Python functions in function_with_rich_values should only be used when the input arguments are independent and the result will not be involved in further operations with any of the original input arguments.

4.1.4. Comparisons between rich values

The RichValue class has special methods for the comparison operators (==, <, >, <=, >=), so they can be compared between them. As rich values represent distributions rather than numbers, confidence intervals are used to define the comparison operations; in particular the result of the interval method is used (see page 8), with the argument sigmas having a value of 1.0 or 3.0, as explained below.[14] That is, it the rich value is a centered value, its 1 σ or 3 σ confidence interval is used, but if it is an upper/lower limit or a constant range of values, its interval of possible values is used instead.

The identity operator (==) determines if two rich values are equivalent, in the sense that their 1 σ confidence intervals overlap. The comparisons of less (<) and greater (>) determine which of the two rich values has the 3 σ confidence intervals with the lowest or greater

14. These default values can be changed, see section 10. Also, you can perform comparisons between rich values using the functions of section 9.4.

edge, respectively. Then, the rest of the operators are the usual combinations between the three that have been explained.

Below there are some examples of comparisons between rich values.

```
x = rv.rval('5.2 +/- 0.3')
y = rv.rval('3.2 +/- 0.3')
z = rv.rval('2.6 +/- 0.9')
print(x > z, x == y, y > z, y == z)
[out]   True False False True
```

Note how it happens that `y` is not greater than `z` (that is, `y > z` is `False`) despite having a greater main value, and also how both variables are equivalent (`y == z`), in the sense that their $1\,\sigma$ confidence intervals overlap.

Having comparison operators defined allows to perform operations such as the maximum and minimum of a group of rich values. For example:

```
print(max(y, z), np.max([x, y, z]))
[out]   2.6+/-0.9  5.2+/-0.3
```

As you can see, one can use NumPy functions like `np.min`, `np.argmin`, `np.argmax` or `np.sort`.

4.1.5. Complex rich values

Rich values consisting of complex numbers are also supported. They are composed of two rich values: the real part and the imaginary part.

It can be created with the `rich_value` function, writing the real and imaginary parts separated by a plus or minus sign (+, −) surrounded by spaces, and putting a `j` surrounding the imaginary part, using brackets if needed; for example, `rv.rval('5.2 +/- 0.3 + (2.8 -0.3+0.4)j')`. This values will be converted into objects of class `ComplexRichValue`. Alternatively, this class can be directly called providing two rich values as an input (or also text strings representing rich values), and optionally the arguments `domain` and `is_int` to replace the corresponding variables of both inputs; for example: `rv.ComplexRichValue('5.2 +/- 0.3', '(2.8 -0.3+0.4)j', domain=[0,inf])`.

Complex rich values have most of the instance variables, attributes and methods of usual rich values, with the addition of the attributes `real` (real part), `imag` (imaginary part), `mod` (module) and `ang` (angle), and also `common_domain` (the union of the domains of the real and imaginary parts).[15] Lastly, operations between complex rich values are supported in the same way as with rich values, that is, using the mathematical operators (`+`, `-`, `*`, `/`, `**`) and using the `function_with_rich_values` function.

4.2. Arrays of rich values

There are three ways to create arrays of rich values within this library: using just NumPy arrays containing rich values, using the function `rich_array`, or using directly the class `RichArray`. We will call these arrays *rich arrays*.

15. Alternative longer names can be used instead (see section 11).

4.2.1. NumPy arrays

Using the functions from the library NumPy (like the function `array`), one can create arrays whose elements are rich values (of class `RichValue`). Below is a simple example of a creation of an array of rich values using the NumPy function `array` and the two ways of creating rich values mentioned in section 2.1.[16]

```
import numpy as np
u = np.array([rv.rval('1.21 +/- 0.14'), rv.rval('< 4')])
u = np.array([rv.RichValue(1.21, 0.14), rv.RichValue(4, is_uplim=True)])
```

With this method, we do not need any additional functions or classes more than those of NumPy. However, we recommend using one of the two other ways of creating arrays of rich values, as they are simpler and more flexible.

4.2.2. Function `rich_array`

It can also be called with the shortened name `rarray`. Below are the arguments of this function, although the first one is the only required:

- **`array`**. It should be a list, an array or similar. The elements of the input argument can be either a rich value (of class `RichValue`) or a text string representing a rich value, that is, with the formatting style explained in section 3.
- **`domain`**. The domain of all the entries of the rich array. If not specified, there are two possibilities: if the entry of the input array is already a rich value, its original domain will be preserved; if not, the default domain $(-\infty, \infty)$ will be used instead.
- **`is_int`**. Logical variable used to indicate that the magnitude corresponding to the rich array is integer. The default is `False`.
- **`use_default_extra_sf_lim`**. This is the same variable as in the function `rich_value` (see section 4.1.1).

For example, the creation of the same array as in the previous example would be, using the shortened name of the function:

```
u = rv.rarray(['1.21 +/- 0.14', '< 4'])
```

In this way, we create an object of class `RichArray`, which is basically the NumPy class `ndarray` with some additional features. Alternatively, we could have created the rich value directly using the class `RichArray`.

4.2.3. Class `RichArray`

This class is inherited from the `ndarray` class from NumPy. An object of this class is basically a NumPy array but with some additional instance variables and methods. It is useful when you already have an array with the central values of a certain variable and another array with the corresponding uncertainties.

Arguments

Below are the arguments of this Python class, being the first element the only required:

16. For the rest of code examples, we will assume that we imported NumPy as `np` (`import numpy as np`).

- **mains**. Array of main values.
- **uncs**. Array of lower and upper uncertainties associated with the main values. It can have the same shape as mains, in which case the uncertainties would be symmetrical, or an array containing lower and upper uncertainties, in a way so that either the first or the last element of the shape of the input array is 2 and corresponds to the lower and upper uncertainties. By default, it is an array of zeros (0).
- **domains**. Array containing the domain for each entry of the rich array. It can also be just one value, which would be applied to all the entries of the resulting rich array. By default, the domain $(-\infty, \infty)$, that is, `[-np.inf, np.inf]`, will be used for all the elements of the resulting rich array.
- **are_lolims**. Array of logical variables that indicate if each entry is actually a lower limit. It can also be just one value, which would be applied to all the entries of the resulting rich array. By default, it is an array of False values.
- **are_uplims**. Array of logical variables that indicate if each entry is actually an upper limit. It can also be just one value, which would be applied to all the entries of the resulting rich array. By default, it is an array of False values.
- **are_ranges**. Array of logical variables that indicate if each entry is actually a finite interval or an upper/lower limit. It can also be just one value, which would be applied to all the entries of the resulting rich array. By default, it is an array of False values.
- **are_ints**. Array of logical variables that indicate if each entry is a rich value of integer nature. By default, it is an array of False values.

As for the argument names for the RichValue class, you can use alternative longer names instead (see section 9). For example, instead of are_uplims you can write are_upper_limits.

As an example, below is a code to create the same rich array as in the previous section:

```
u = rv.RichArray([1.21,4], [0.14,0], are_uplims=[False,True])
```

Instance variables

The RichArray class does not have any instance variable. Instead, you can use the attributes and some of the methods to access to some of the properties of the entries of the rich array.

Attributes

The RichArray class have several attributes, which are basically the implementations of the instance variables and attributes of the RichValue class. Below is the list of all of them.

- **mains**. Main values of the elements of the rich array.
- **uncs**. Uncertainties of each element of the rich array.
- **domains**. Array containing the domain of each entry of the rich array.
- **are_lolims**. Array of logical variables that describe if each element of the rich array is a lower limit.
- **are_uplims**. Array of logical variables that describe if each element of the rich array is

an upper limit.

- **are_ranges**. Array of logical variables that describe if each element of the rich array is a constant range of values.
- **are_ints**. Array of logical variables that describe if each element of the rich array is a constant range of values.
- **nums_sf**. Array containing the number of significant figures of each entry of the rich array.
- **min_exps**. Array containing the minimum exponent to use scientific notation of each entry of the rich array.
- **max_decs**. Array containing the maximum number of decimals to be displayed of each entry of the rich array.
- **extra_sf_lims**. Array containing the limit for extra significant figre of each entry of the rich array.
- **are_lims**. Array if logical variables that describe if each element of the rich array is an upper/lower limit.
- **are_intervs**. Array of logical variables that describe if each element of the rich array is a constant range of values or an upper/lower limit.
- **are_centrs**. Array of logical variables that describe if each element of the rich array is a centered value (a main value with uncertainty).
- **are_exacts**. Array of logical variables that describe if each element of the rich array is an exact value (a centered value with zero uncertainty).
- **are_consts**. Array of logical variables that describe if each element of the rich array is a centered value (an exact value with a domain consisting of just the main value).
- **are_finites**. Array of logical variables that describe if each element of the rich array corresponds to a finite value (centered value or finite range).
- **are_infs**. Array of logical variables that describe if each element of the rich array correspond to a infinite interval.
- **are_nans**. Array of logical variables that describe if each element of the rich array is a NaN (*not a number*).
- **centers**. Central values of the elements of the rich array that are centered values.
- **uncs_eb.** Uncertainties of each element of the rich array as an array where the first element of its shape is 2 and corresponds to the lower and upper uncertainties, so that it can be easily used with Matplotlib's function `errorbar`.[17]
- **rel_uncs**. Relative uncertainties of each element of the rich array.
- **signals_noises**. Signal-to-noise ratios (SN) for each element of the rich array.
- **ampls**. Distances between the central value and the bounds of the domain (which we call *amplitudes*), for each element of the rich array.

17. You can instead use the function `errorbar` within this library (see section 6).

- **rel_ampls**. Distances between the central value and the bounds of the domain divided by the uncertainties, for each element of the rich array (*relative amplitudes*).
- **norm_uncs**. Uncertainties divided by the distances between the central value and the bounds of the domain, for each element of the rich array (*normalized uncertainties*).
- **prop_scores**. Array with the *propagation score* (result of the RichValue class method prop_score()) for each entry of the rich array.
- **datatype**. Type of number of the rich values of the rich array (int, float or complex) needed to describe all of the entries.
- **variables**. List of independent variables (non-correlated) of the rich array, that is, the union of the variables attribute for each entry of the rich array.
- **expression**. Expression representing the rich value in terms of its independent variables.

Like for the instance variable and attribute names for the RichValue class, you can use alternative longer names instead (see section 11). For example, instead of norm_uncs you can write normalized_uncertainties.

Methods

The RichArray class have several proper methods, apart from the ones inherited by Numpy arrays. Below is a list of the proper ones:

- **intervals**(self, sigmas). Array with the result of the RichValue method interval(self, sigmas) for each entry of the rich array; by default, sigmas = 3.0.
- **signs**(self, sigmas). Array with the result of the RichValue method sign(self, sigmas) for each entry of the rich array; by default, sigmas = 3.0.
- **medians**(self, num_points). Array with the result of the RichValue method median (self, num_points) for each entry of the rich array; by default, num_points = int(1e4).
- **means**(self, num_points). Array with the result of the RichValue method mean (self, num_points, sigmas) for each entry of the rich array; by default, num_points = int(1e4) and sigmas = 8.0.
- **modes**(self, num_points). Array with the result of the RichValue method mode (self, num_points, sigmas) for each entry of the rich array; by default, num_points = int(1e4) and sigmas = 8.0.
- **variances**(self, num_points). Array with the result of the RichValue method var (self, num_points, sigmas) for each entry of the rich array; by default, num_points = int(1e4) and sigmas = 8.0.
- **stds**(self, num_points). Array with the result of the RichValue method std (self, num_points, sigmas) for each entry of the rich array; by default, num_points = int(1e4) and sigmas = 8.0.
- **moments**(self, num_points). Array with the result of the RichValue method moment (self, num_points, sigmas) for each entry of the rich array; by default, num_points = int(1e4) and sigmas = 8.0.

- **latex**(self, dollars, mult_symbol). Array of text strings of a LaTeX code representing each entry of the rich array, using mult_symbol as the multiplication symbol in scientific notation (by default, mult_symbol = \\cdot). The logical variable dollars, which is True by default, determines if the text string is enclosed with dollar symbols ($) or not.
- **set_params**(self, params). It modifies the parameters of the entries of the rich array specified in the params variable (domain, num_sf, min_exp, max_dec, or extra_sf_lim), which must be a dictionary containing entries with the name of each variable to be set and the corresponding desired values.
- **set_lims_uncs**(self, factor). For each element of the rich array, if the rich value is an upper/lower limit, the lower/upper uncertainty (stored in the instance variable unc) will be defined as the central value divided by the factor (by default, factor = 4.0). If two values are provided in the variable factor, the first one will be user for lower limits and the second one for upper limits. This can be useful when plotting upper/lower limits with Matplotlib's function errorbar.[17]
- **sample**(self, len_sample). For each entry of the rich array, it returns a sample of size len_sample of the distribution associated with the corresponding rich value. By default, len_sample is 8 000.
- **function**(self, function, **kwargs). It applies the given function (function) to every element of the rich array. Additional arguments can be passed (**kwargs), which are the arguments of the function function with_rich_arrays (see next section).

The last method allows to apply a function to the rich array and therefore to obtain a new one. See next section for more details on making operations with rich arrays.

4.2.4. Operations with arrays of rich values

The RichArray class inherits the methods for the basic arithmetic operations from the ndarray and RichValue classes. Therefore, you can just use the arithmetic operators (+, -, *, /, **, //, %) to combine a rich array with another rich array, another rich value or a usual number.

The uncertainties will be propagated automatically, and the condition of an upper/lower limit (or even a finite interval) will be taken into account to obtain the final result. Also, correlation between variables will be taken into account. For example:

```
u = rv.rarray(['1.2 +/- 0.4', '5.8 +/-0.9'])
v = rv.rarray(['8 +/- 3', '< 21'])
u * v
[out]   RichArray([9-4+5, < 150], dtype=object)

x = rv.rval('2.0 +/- 0.3')
u + x
[out]   RichArray([3.2+/-0.5, 7.8+/-0.9], dtype=object)
```

If instead of working with objects of class RichArray you work with NumPy arrays (class

ndarray) whose elements are rich values (class RichValue), you can perform this kind of operations as well.

Additionally, you can use the function function_with_rich_arrays, which is like function_with_rich_values but for rich arrays. Additionally, there is another function that you can use to apply a custom mean to a rich array, called fmean. Both functions are explained below.

Function function_with_rich_arrays

Using this function one can compute any expression involving rich arrays, which can be element by element or not. This function can also be called with the shortened name array_function. Its arguments are the same as function_with_rich_values plus an additional argument, elementwise. Below are the most important ones:

- **function**. Function to be applied to the input rich arrays. It should be a text string representing the source code of the function (see function_with_rich_arrays for more details), although it can also be a Python function. If the function is not to be applied element-wise, it must be a Python function, and in this case it can return several numeric values. Therefore, the output itself can be a new array; however, it could not be consisted of several arrays.
- **args**. List with the input rich arrays, in the same order as the arguments of the given function.
- **len_samples**. Size of the samples of the elements of the arguments that will be drawn for calculating the final distribution applying the input function. The default is the square root of the number of arguments times the default sample size (8 000).
- **elementwise**. Logical variable that states if the given Python function has to be applied element-wise for each of the elements of the input rich arrays. By default it is False.

  Let's see some examples of the use of this function, using its shortened name.

```
u = rv.rarray(['1.2 +/- 0.4', '5.8 +/-0.9'])
v = rv.rarray(['8 +/- 3', '< 21'])
rv.array_function('{}*{}, [u,v], elementwise=True)
[out]   RichArray([9-4+5, < 150], dtype=object)
```

That would be it. In this case, the product of two NumPy arrays is defined element-wise, so it is not necessary to specify that the given function has to be applied element by element. The usage for a function which is not element-wise, like the scalar product, would be very similar, but now we should pass the Python function directly.

```
rv.array_function(np.dot, [u,v])
[out]   < 160
```

Finally, we can also apply a function that returns a new array as an output, like the cross product.

```
u = rv.rarray(['3.0 +/- 0.4',  '2.1 +/- 0.3', '0.0 +/-0.3'])
v = rv.rarray(['6.4 +/- 0.8', '-3.6 +/- 0.4', '0.0 +/-0.2'])
rv.array_function(np.cross, [u,v])
[out]   RichArray([0-1.2+1.1, 0+/-2.0, -24+/-3], dtype=object)
```

Lastly, bear in mind that this function can be used to apply operations to rich arrays and rich values, as the last ones will be converted to rich arrays with zero dimension (size 0).

Function fmean

This function applies a generalized quasi-arithmetic mean (or *f*-mean) along all the elements of the input array, which can be a rich array.[18] In order to use it, you have to specify the function to be used for the mean and its inverse. Below is the list of its arguments:

- **array**. Input array to apply the mean.
- **function**. Function that defines the mean, as a Python function or a text string representing it. By default, it is the identity (which corresponds to the arithmetic mean).
- **inverse_function**. Inverse of the function that defines the mean, as a Python function. By default, it is the identity (which corresponds to the arithmetic mean).
- **weights**. Weights to be applied to the values of the input array. By default, they are equal weights.
- **weight_function**. Function to be applied to the weights before normalization. By default, it is the identity (no operation is applied).

Besides these arguments, you can write more arguments of the function function_with_rich_arrays (or, equivalently, of function_with_rich_values). Also, there is an additional function called mean that is like fmean but just applies the arithmetic mean (so the arguments function and inverse_function are missing).

Let's see an example of the use of this function for calculating the arithmetic mean.

```
u = rv.rarray(['1.2 +/- 0.4', '5.8 +/-0.9'])
rv.fmean(u)
[out]   3.5+/-0.5
```

Alternatively, we could have used rv.mean isntead of rv.fmean. And now, the same example but with the geometric mean.

```
u = rv.rarray(['1.2 +/- 0.4', '5.8 +/-0.9'], domain=[0,np.inf])
rv.fmean(u, function=np.log, inverse_function=np.exp)
[out]   2.6+/-0.5
```

4.3. Tables of rich values

With this library one can also create tables of rich values, that is, groups of rich values labeled with a column name and an index. This is done using Pandas dataframes, so we will call this objects *rich dataframes*. There are three ways of creating this kind of object: with Pandas dataframes, with the function rich_dataframe, and with the class RichDataFrame.

4.3.1. Pandas dataframes

Using the library Pandas, one can create dataframes whose elements are rich values (of class

18. If f is a function with an inverse function f^{-1}, $\{x_i\}$ is a set of input numbers, and w_i are a set of weights for the input numbers, the generalized *f*-mean of $\{x_i\}$ is: $\langle x \rangle = f^{-1}\left(\frac{1}{\sum_i w_i} \sum_i w_i f(x_i)\right)$.

RichValue). Below is a simple example of a creation of a dataframe of rich values using the Pandas class DataFrame.[19]

```
import pandas as pd
arr = rv.rarray([['2.1+/-0.3','3.4+/-0.4','<4'],['5','<6','8+/-1']])
df = pd.DataFrame(arr, columns=['a','b','c'])
```

We could have created the dataframe from a Python dictionary as well.[20]

```
dic = {'a': rv.rarray(['2.1+/-0.3','5']),
       'b': rv.rarray(['3.4+/-0.4','<6']),
       'c': rv.rarray(['<4','8+/-1'])}
df = pd.DataFrame(dic)
```

With this method, we do not need any additional functions or classes more than those of Pandas. However, we recommend using one of the two other ways of creating dataframes of rich values, as they are simpler and are more flexible.

4.3.2. Function rich_dataframe

This function converts an input dataframe containing rich values or text strings representing rich values to a rich dataframe. It can also be called with the shortened names rich_df or rdataframe. Below are the arguments of this function, although the first one is the only required:

- **df**. It should be a dataframe, whose elements can be either a rich value (of class RichValue), a text string representing a rich value (with the formatting style explained in section 2.1.1), or a non-numeric text string; in this last case, the text string will be preserved as it is.
- **domains**. Dictionary containing, for each column, the domain of its elements, that is, the minimum and maximum values that the magnitude corresponding to each column could take. Instead, a common domain for all the columns can be directly specified. If not specified, there are two possibilities: if the entry of the input dataframe is already a rich value, its original domain will be preserved; if not, the default domain $(-\infty, \infty)$ will be used instead.
- **use_default_extra_sf_lim**. This is the same variable as in the function rich_value (see section 4.1.1).

Additionally, you can use keyword arguments from Pandas' DataFrame class.

As an example, below is a code to create a simple dataframe from either an array-like list or a dictionary, as in the previous section.

```
arr = [['2.1+/-0.3','3.4+/-0.4','<4'], ['5','<6','8+/-1']]
rdf = rv.rich_dataframe(arr, columns=['a','b','c'])
dic = {'a': ['2.1+/-0.3','5'], 'b': ['3.4+/-0.4','<6'],
       'c': ['<4','8+/-1']}
```

19. For the rest of code examples, we will assume that we imported Pandas as pd (import pandas as pd).

20. A Python dictionary is an object than includes several variables identified by keywords. To create one, you have to write the pairs of keywords and variables (name of the keyword, a colon, and the variable) separated by commas, and all of this enclosed by keys; for example: dic = {'a': 1, 'b': 2}.

```
rdf = rv.rich_dataframe(dic)
```

As you can see, this method is easier and faster than the one of the previous section. Also, if any element of the input list/array/dictionary for creating the dataframe is a non-numeric text string, it will be preserved to the final rich dataframe. In this way, we create an object of class RichDataFrame, which is basically the Pandas class DataFrame with some additional features. Alternatively, we could have created the rich dataframe using the class RichDataFrame, although it is always easier and more practical to use the function rich_dataframe.

4.3.3. Class RichDataFrame

This class is inherited from the DataFrame class from Pandas. An object of this class is basically a Pandas dataframe but with some additional methods. It also uses a supplementary class, RichSeries, which is basically a Pandas Series class but with the proper attributes from the RichArray class.

Argument

The only argument required for creating an object of this class is a dataframe. This class just adds several methods to the input dataframe, but in order to work as expected all the entries of the input dataframe should be either rich values or text strings. Therefore, instead of calling to this class for creating a rich dataframe, it is better to use the rich_dataframe function explained above.

Attributes

The RichDataFrame class has all the DataFrame attributes, that allow to visualize and modify its values.

Basically, its attributes correspond to properties of rich values, such as mains, uncs, ampls, etc. They return a regular DataFrame object containing the result of applying the corresponding RichArray method to the values of the rich dataframe, with the exception of the text strings, which will be preserved.

For example, if rdf is a RichDataFrame object, rdf.mains will return a regular dataframe (DataFrame) whose entries will be the main values of the rich values of the rich dataframe, preserving all the entries which are just text strings; similarly, rdf.uncs would return a regular dataframe in which, in every entry of the original rich dataframe containing a rich value, there would be a two-element list containing the inferior and superior uncertainty of that rich value.

Methods

The RichDataFrame class has all the DataFrame methods and also has some proper methods that allows to visualize and modify the rich dataframe. These proper ones are listed below:

- **flatten_property**(self, name). This can be used to modify the resulting data-frame of applying any of the implemented RichArray attributes and methods that correspond to properties of rich values that are two-element lists (such as uncs, are_uplims, ampls, intervals(), etc.). It applies the input attribute/method name (name), which should be

the name of the attribute/method as a text string, to the current rich dataframe, including the brackets and possible arguments in the case of methods. Then, if it is an attribute/method that returns two values for each of the rich value entries (like uncs, which returns the inferior and superior uncertainties), the output will be a list containing two dataframes, each of them containing one of the attribute results. For example, if name = 'uncs', the output will be a list with two dataframes, one containing the inferior uncertainties of the present rich values and another one containing its superior uncertainties.

- **set_params**(self, params). It modifies the parameters of each entry of the specified parameters in the params variable, containing entries with the name of the variables to be modified (domain, num_sf, min_exp, and extra_sf_lim). The value of each entry must be another dictionary containing one entry for each column of the dataframe that will be modified, with the corresponding desired value.
- **latex**(self, return_df, export_frame, export_index, row_sep, show_dollars, mult_symbol). If return_df = True, it returns a dataframe of text strings of a LaTeX code representing each entry of the original dataframe, using mult_symbol as the multiplication symbol in scientific notation (by default, mult_symbol = \\cdot). Instead, if return_df = False (by default), it returns a text string in the LaTeX formatting style representing the content of a table, using row_sep as the text that indicates the end of a row (by default, row_sep = \\tabularnewline). If export_frame = True (by default), the output will include the tabular lines that create the frame of the table, including also the header with the names of the columns. If export_index = True (by default), the index of the dataframe will be also exported. The logical variable show_dollars, which is True by default, determines if the text strings for each element are enclosed with dollar symbols ($) or not.
- **set_lims_uncs**(self, factors). For each element of each column of the dataframe, if the rich value is an upper/lower limit, the uncertainty instance variable (unc) will be replaced as the central value divided by the factor specified for each column by the variable factors (by default, it is 4.0 for every column). If two values are provided in the variable for a certain column, the first one will be user for lower limits and the second one for upper limits. This can be useful when plotting upper/lower limits with Matplotlib's function errorbar.[9]
- **create_column**(self, function, columns, **kwargs). It returns a new column of type rich array obtained applying the given function (function) to the specified columns (columns) of the dataframe. The rest of the arguments (kwargs) are the same as in funtion_with_rich_values.
- **create_row**(self, function, rows, **kwargs). It returns a new row obtained applying the given function (function) to the specified rows (rows) of the dataframe. The rest of the arguments (kwargs) are the same as in function_with_rich_values.

The last two methods allow to apply a function to the rich dataframe and obtain a new column or row, which can then be added to the dataframe, as can be seen in the example of the

next section.

4.3.4. Operations with rich tables of rich values

Simple operations can be done with rich dataframes using the arithmetic operators (+, -, *, /, **) to combine a rich dataframe with another rich dataframe, another rich value or a usual number. The uncertainties will be propagated automatically, and the condition of an upper/lower limit (or even a finite interval) will be taken into account to obtain the final result. Also, correlation between variables will be taken into account. Note that this basic operations will only work if all the entries of the dataframe are numeric.

For operations with more complex functions, you should use the methods `create_column` and `create_rows`. They are also useful if you want to make operations with the data within the dataframe.

Below is an example of how to create a new column in a dataframe combining its columns with the method `create_column`.

```
dic = {'a': ['6.4+/-0.5','8'], 'b': ['3.4+/-0.4','<6'],
       'c':['4','8+/-1']}
rdf = rv.rich_dataframe(dic, domains=[0,np.inf])
rdf['d'] = rdf.create_column('{}/{}+{}', ['a','b','c'])
```

4.3.5. Pandas series with rich values

A Pandas series is a one-dimensional array with an index array; in fact, when extracting a column or a row from a Pandas dataframe, the result is a series. In order to directly create series that contain rich values (*rich series*), we can use the function `rich_series`, which takes a Pandas series containing text strings representing rich values as an input and has the same optional arguments as the function `rich_array`. Alternatively, if the series already contains rich values, we can use the method `RichSeries`. Otherwise, when extracting a column or a row from a rich dataframe, the result will be a rich series.

5. Importing and exporting of rich values

The importing and exporting of rich values is very simple.

5.1. Importing

For the importing from a plain text file, you can write text strings representing rich values, with the formatting style explained in section 2.1.1. Then, any function that can import text strings will be fine, like NumPy's `loadtxt` or Pandas' `read_csv`. For example, below is a simple table from a file in `.csv` that could be imported into a rich dataframe.

Source \ Molecule	HC3N	CH3CN
L1517B	5.16+/-0.03 e13	2.1+/-0.3 e11
L1498	1.6+/-0.3 e13	< 8.3 e10
L1544	1.0+/-0.3 e14	1.5+/-0.2 e11
B1-a	4.2+/-1.2 e12	4.9+/-1.1 e11

B1-c	4.2+/-3.4 e12	3.5+/-0.6 e11
SVS 4-5	1.1+/-0.3 e13	5.2+/-0.9 e11
GM Aur	1.9-0.4+0.4 e13	2.1-0.1+0.2 e12
As 209	2.9-0.5+0.5 e13	1.7-0.2+0.2 e12
HD 163296	7.3-1.9+2.5 e13	2.3-0.2+0.2 e12
MWC 480	7.8-2.7+3.9 e13	3.5-0.2+0.2 e12
# (units: /cm2)		
46P	< 0.003	0.017+/-0.001
67P	0.0004	0.0059
# (units: % /H2O)		

It contains the abundances of two molecules (HC_3N and CH_3CN) in different astronomical objects. Assuming that the file is named table.csv, we could import the data as:

```
df = pd.read_csv(table.csv, index_col=0, comment='#')
rdf = rv.rich_dataframe(df, domain=[0,np.inf])
```

As the abundances can only be positive, we specified a domain of `[0, np.inf]`; alternatively, we could have specified the domain in the first entry of each numeric column, as the function `rich_dataframe` will take, for each column, the domain of the first rich value as the domain of the whole column; that is, we should have written `5.16+/-0.03 e13 [0,inf]` and `2.1+/-0.3 e13 [0,inf]` as the first value of the columns labeled as HC3N and CH3CN, respectively. The result of the importing will be a rich dataframe whose indexes are the first original column of the table in the .csv file, and with two columns with numeric values named HC3N and CH3CN.

5.2. Exporting

For the exporting of a variable containing rich values to a plain text file, you can use any function that can export text strings, like NumPy's `savetxt` or Pandas' `to_csv` (which is a method for the class DataFrame). For example, below is a code for adding a new column to the example table of the previous section as the ratio between the two numeric columns, and exporting it to a .csv file.

```
rdf['ratio'] = df['HC3N'] / df['CH3CN']
rdf.to_csv(table-ratio.csv)
```

And the resulting file would look like the table below.

Source \ Molecule	HC3N	CH3CN	ratio
L1517B	5.16+/-0.03 e13	2.1+/-0.3 e11	240-30+40
L1498	1.6+/-0.3 e13	< 8 e10	> 100
L1544	1.0+/-0.3 e14	1.50+/-0.20 e11	670-210+230
B1-a	4.2+/-1.2 e12	4.9+/-1.1 e11	8-3+4
B1-c	4+/-3 e12	3.5+/-0.6 e11	12-9+11
SVS 4-5	1.1+/-0.3 e13	5.2+/-0.9 e11	21-6+8
GM Aur	1.9+/-0.4 e13	2.10-0.10+0.20 e12	8.9-1.9+2.0
As 209	2.9+/-0.5 e13	1.70+/-0.20 e12	17-3+4
HD 163296	7.3-1.9+2.5 e13	2.30+/-0.20 e12	32-8+11
MWC 480	8-3+4 e13	3.50+/-0.20 e12	22-8+11
46P	< 3 e-3	1.70+/-0.10 e-2	< 0.21
67P	4.0 e-4	5.9 e-3	0.068

We lost the comments (starting with #) of the original table, but we could actually have added it adding to the dataframe a row with the comment as the index and an empty string (`''`) as the value for each column. Or, alternatively, we could just add the comments directly to the resulting .csv file.

6. Plotting rich values

In order to easily plot arrays or lists of rich values, this library offers the function `errorbar`, which is basically an implementation of Matplotlib's `errorbar` function.

Function `errorbar`

It accepts arrays of rich values (which can be rich arrays or not) as inputs, as well as all the keyword arguments of Matplotlib's same-name funtion. Here are the specific arguments of this function:

- **`x`**. Array of rich values that will be plotted on the horizontal axis.
- **`y`**. Array of rich values that will be plotted on the vertical axis.
- **`lims_factor`**. List containing the two factors that define the sizes of the arrows for displaying the upper and lower limits, respectively (the bigger the factor, the smaller the arrow will be). It can be just one value, that will be used for both limits. This is an optional argument; by default the factors will be calculated automatically.

The rest of the arguments are the keyword arguments for Matplotlib's `errorbar` function.[21] As an output, it returns the same object as in Matplotlib: an `ErrorbarContainer` object. Therefore, additional functions like `xlabel`, `xlim`, `title`, etc, can be used to tweak the appearance of the plot; and working with `axes` objects is also supported.

Values with a central value and uncertainties will be plotted as points with error bars; upper/lower limits will be plotted as a point (the limit value) and an arrow; and constant ranges of values will be plotted as just an error bar with no point. By default, the color of the points will be gray, and the color of the error bars and arrows will be black.

The use of Matplotlib's `errorbar` function arguments `capsize` and `capthick` are discouraged for many reasons. First of all, errorbar caps are used when displaying a point $(x_i,\ y_i)$ where one of the variables is a constant range of values and the other has uncertainties, to show the uncertainty of that variable; therefore, their use for centered values could be misleading. Secondly, there is a bug that adds a little errorbar to upper/lower limits when using any of these two arguments. Finally, caps can create a plot too baroque, specially if the pair of values $(x_i,\ y_i)$ have both uncertainties.

Let's see a quick example of how to use this function. Consider the following table containing two columns of rich values (split in two halves for a better visualization):

x	y	x	y
< 20	6 +/- 4	19.4 +/- 1.0	47 +/- 4
2.7 +/- 1.1	16 +/- 3	20.7 +/- 0.8	55 +/- 3

21. See Matplotlib's documentation: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.errorbar.html .

3.4 +/- 1.2	16 +/- 4	20.8 +/- 0.6	56 +/- 4
4.4 +/- 0.7	15 +/- 3	21.4 +/- 0.9	47 +/- 3
8.6 +/- 0.9	30 +/- 4	21.8 +/- 0.7	55 +/- 4
8.7 +/- 1.2	30 +/- 4	25.7 +/- 1.0	< 100
10.0 +/- 0.9	32 +/- 4	28.5 +/- 0.8	69 +/- 3
11.1 +/- 1.5	36 +/- 3	29.3 +/- 0.9	66 +/- 3
11.9 +/- 0.8	42 +/- 4	35.3 +/- 1.4	84 +/- 3
13.3 +/- 0.7	36 +/- 5	35.8 +/- 1.0	78 +/- 3
14.2 +/- 0.6	31 +/- 3	36.0 +/- 1.2	83.1 +/- 1.7
14.8 +/- 0.7	42 +/- 6	38.5 +/- 0.9	81 +/- 3
16.1 +/- 1.2	42 +/- 6	39.0 +/- 0.9	86 +/- 6
16.5 +/- 0.8	46 +/- 4	39.2 +/- 0.9	90 +/- 4
17.9 +/- 0.6	45 +/- 3	42.5 +/- 1.1	47 -- 99

Supposing that this table is stored in a .csv file named table.csv, we could import it with Panda's read_csv function, and then just plot both columns with errorbar.

```
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv(table.csv)
rdf = rv.rich_dataframe(df)
plt.figure(1, figsize=(7,4))
rv.errorbar(rdf['x'], rdf['y'], color='gray')
plt.xlim(left=0)
plt.ylim(bottom=0)
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.tight_layout()
```

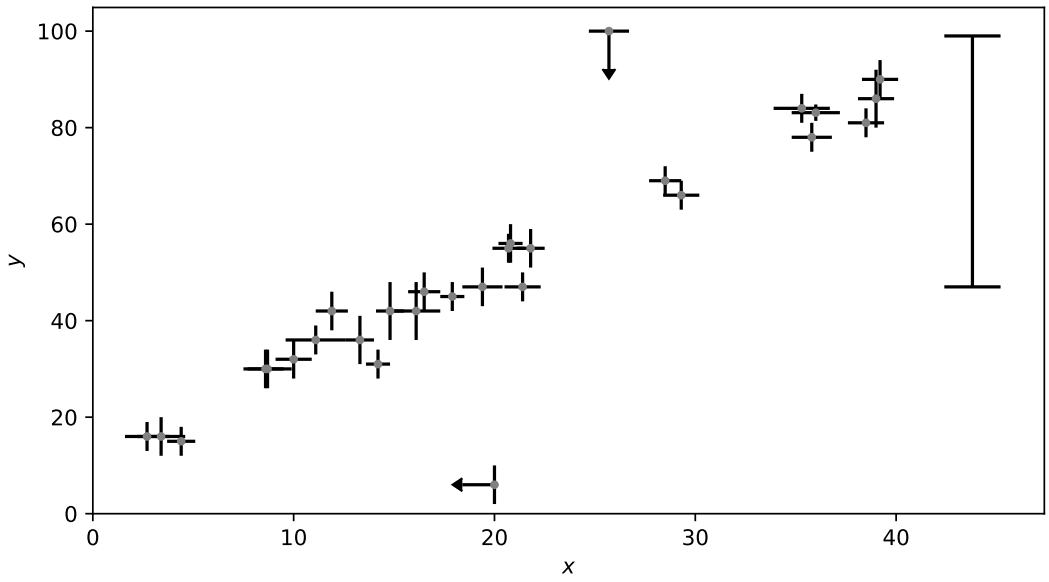

Figure 1. Plot of the example data shown in the table above using the function errorbar.

Figure 1 shows the resulting plot. As you can see, making the plot itself is very simple (just a call to the errorbar function). The colors of the points and the errorbars can be modified using the corresponding keyword arguments from Matplotlib's errorbar function.

7. Making fits with rich values

Performing a fit of a set of rich values to a given model can be easily done with some functions included in this library: `point_fit` and `curve_fit`.

They rely on SciPy's function `minimize`, which will minimize a loss (error) function between several samples of the input rich values and the predictions given by the model function; by default, the loss function will be the mean squared error (MSE).

In order to take into account for the status of rich value of the input (presence of uncertainties, upper/lower limits, and/or constant range of values), and also to obtain the fitted parameters as rich values (obtaining a central value and uncertainties in most cases), the optimization process would be repeated in several iterations. In each one, one value will be sampled for each input rich value and the fit will be performed with these values. Therefore, the result would be a distribution of values for each parameter of the model function, which will be represented with a rich value for each of them.

Function `curve_fit`

It makes a fit of two arrays of rich values, $\vec{x}$ and $\vec{y}$, treating the second one as dependent of the first, and given a function that converts the independent variable into the dependent one. That is, it performs a fit of $\vec{y}$ over $\vec{x}$ with respect to a model function $f(x, \vec{\theta})$, that depends on the dependent variable, x, and a certain set of parameters, $\vec{\theta}$, and that applied to a sample of the input array $\vec{x}$ returns an array $\vec{y}$', which are the predictions of the input array $\vec{y}$.[22]

In order to compute the loss, only the rich values of $\vec{x}$ that are not intervals (that is, they have a central value and uncertainties, even if they are zero) are sampled to then make the predictions of $\vec{y}$.[23] Then, the mean loss between a sample of $\vec{y}$ and its predictions, $\vec{y}$', is computed. Finally, it is checked that the rich values of $\vec{x}$ that are actually intervals are consistent with the current model.[19] Additionally, the real dispersion between the original values of the dependent variable ($\vec{y}$) and the modeled ones ($\vec{y}$') is estimated.

Below are the arguments of this function:

- **`x`**. Input array of rich values corresponding to the independent variable.
- **`y`**. Input array of rich values corresponding to the dependent variable.
- **`function`**. Python function whose parameters will be optimized with respect to the given rich values. Its first argument has to be the independent variable (x), and then the parameters of the model ($\vec{\theta}$).
- **`guess`**. List containing an starting value for each of the parameters of the given function ($\vec{\theta}$). These values will be used to start the optimization process.
- **`num_samples`**. Number of different samples of the input rich values that will be drawn. It will be the number of times that the fit will be performed to obtain a distribution of values for each parameter. By default, it is `3000`.

22. Both the input rich values, $\vec{y}$, and the model parameters, $\vec{\theta}$, can have just one element. However, to obtain a unique solution the number of parameters should be less or equal than the number of input rich values.

23. Unless the argument `use_easy_sampling` is set to `True`.

- **loss**. Python function that defines the error between a rich value and a prediction of it. The function to be minimized will be the mean error between the input rich values and the predictions of the model function. By default, it is the squared error (as explained at the beginning of the section).
- **lim_loss_factor**. Factor to enlarge the loss if the rich value is not a centered value and the prediction falls outside the interval of possible values of the rich value. By default it is `4.0`.
- **consider_arg_intervs**. Logical variable that determines if upper/lower limits and constant ranges of values in the input data are taken into account during the fit. This option increases considerably the computation time. Therefore, by default it is `False`.
- **consider_param_intervs**. Logical variable that determines if upper/lower limits and constant ranges of values are be taken into account for calculating the fit parameters. The default is `True`.
- **use_easy_sampling**. Logical variable that determines how upper/lower limits and constant ranges of values are sampled during the fitting. If `True`, upper/lower limits and constant ranges of values will be sampled as usual, with uniform distributions for finite intervals of values and lognormal distributions for infinite intervals. If `False`, intervals will not be sampled for the fitting itself, but will be taken into account for calculating the loss function for the fit, as long as `consider_intervs` is `True`. By default it is `False`.

Besides, additional keyword arguments from SciPy's function `minimize` can be specified.

The output of the function will be a Python dictionary containing the results of the optimization process, like with the function `point_fit`. Its entries are the following:

- **parameters**. List containing the fitted parameters ($\vec{\theta}$) as rich values.
- **dispersion**. Estimated real dispersion between the original values of the dependent variable ($\vec{y}$) and the modeled ones ($\vec{y}$'). Only the centered values are used for this calculation, but taking into account its uncertainties.
- **loss**. Final mean loss between the original points ($\vec{y}$) and the modeled ones ($\vec{y}$').
- **parameters samples**. Array containing the samples of the fitted parameters used to compute the rich values.
- **dispersion sample**. Sample of the estimated real dispersion between $\vec{y}$ and $\vec{y}$'.
- **loss sample**. Array containing the loss corresponding of each group of fitted parameters in the `samples` entry.
- **number of fails**. Number of times that the fit failed, for the iterations among the different samples (the total number of iterations is equal to `num_samples`).

Now, let's see an example of the use of this function. Consider the same data than in figure 1 (section 6). It seems that the data follow a linear trend. Therefore, a linear function could be used to model it: $f(x\,;m,b) = m{\cdot}x + b$, with m being the slope and b the offset We will use the values (m, b) = (2.0, 10.0) as a first guess done by eye. Again, we suppose that the input data are stored in a file named `table.csv`.

```
import pandas as pd
```

```
df = pd.read_csv(table.csv)
rdf = rv.rich_dataframe(df)
function = lambda x,m,b: m*x + b
result = rv.curve_fit(rdf['x'], rdf['y'], function,
                      guess=[2.,10.], consider_intervs=True)
result['parameters']
[out]   [1.94+/-0.07, 12.0+/-1.5]
```

The optimization process should take a few moments, less than a minute in any case. This computation time can be decreased disabling the option `consider_intervs`, which in this particular case leads to virtually the same results. Using the `samples` entry from the output dictionary, we can plot a sample of the fitted models and also the median model (that is, the one with the median slope and median offset), as in figure 2.[24]

Function `point_fit`

It makes a fit of an input set of rich values, $\vec{y}$, with respect to a given function, $f(\vec{\theta})$, that depends on a certain set of parameters, $\vec{\theta}$, and that makes a prediction of the input rich values, $\vec{y}' = f(\vec{\theta})$.[25]

The arguments and the output of this function are the same as the `curve_fit` function explained above, with the only exception that here there is not any independent variable (x), so the argument `x` is not present.

Let's see a quick example of how to use this function. Consider the following set of rich values: (5.8 ± 0.6, 2.2 ± 0.4, 0.43 ± 0.08). Now suppose that we know that these three values can be modeled with the function $f(a,b) = (a^2 + b,\ a^2 - b,\ 2\,b\,/a^2)$. We need to make a first

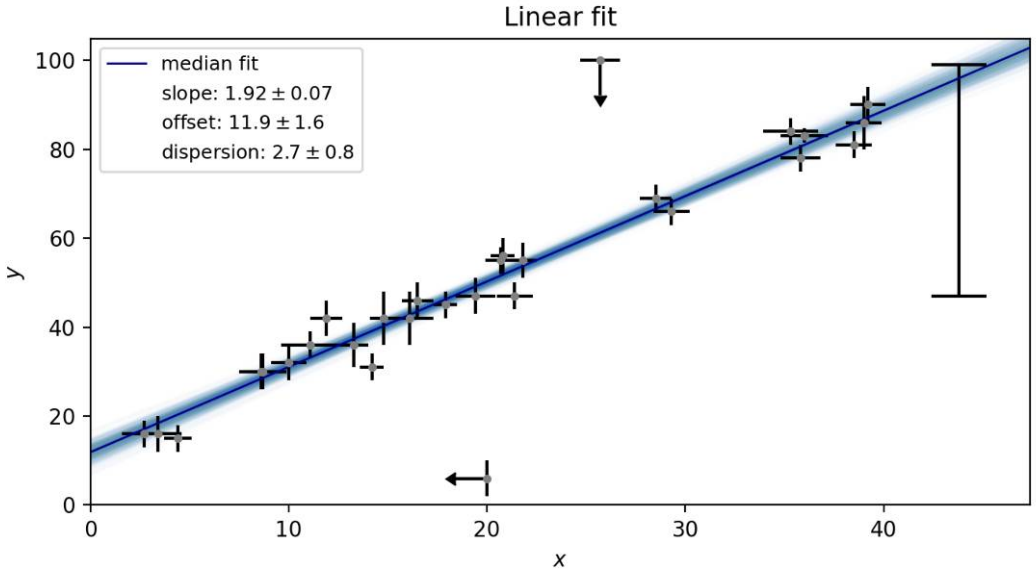

Figure 2. Plot of the example data shown in the table from section 6 (figure 1) and the linear fit performed in the code above. The fitted parameters and the estimated real dispersion of the data are also shown.

24. The code to obtain such a plot can be seen in the example script `linearfit.py`, in the `examples` folder of the GitHub repository: https://github.com/andresmegias/richvalues/ .
25. Both the input rich values, $\vec{y}$, and the model parameters, $\vec{\theta}$, can have just one element. However, to obtain a unique solution the number of parameters should be less or equal than the number of input rich values.

guess of the parameters, which, by trial and error, could be something like $a = 2.0$, $b = 1.2$.[26] Then, the fitting would be quite simple.

```
y = rv.rarray(['5.4 +/- 0.6', '2.6 +/-0.4', '0.83 +/- 0.08'])
function = lambda a,b: (a**2 + b, a**2 - b, 2*b / a**2)
result = rv.point_fit(y, function, guess=[2.0,1.2])
result['parameters']
[out]   [2.00+/-0.09, 1.4+/-0.3]
```

The optimization should take just a few seconds, and the results are shown in Figure 2.

8. Mathematical basis of operations with rich values

Here is explained the algorithm used to make the operations between rich values, that is, to apply a certain function f to a group of rich values. Let's consider a group of n rich values x_i, which can be a value with a central value μ_i and lower and upper uncertainties $(\sigma_{i,1}, \sigma_{i,2})$, an upper/lower limit, or even a finite interval. Each rich value has a probability density function (PDF) associated with it, depending on the properties of the rich value.

As each rich value has a PDF, for each of them we can draw a sample of a large number of values (several thousands), obtaining a set of n distributions $\{x_i\}$; or, grouped different, we have m groups of n values, each group containing a specific value of each variable x_i. Then, we can apply the function f to each of the m groups of values, obtaining a new distribution of values, $\{f(\{x_i\})\}$. Lastly, we have to determine if the distribution is localized around a certain value, in which case it would represent a rich value with a central value and lower and upper uncertainties, of it is more sparse, in which case it would an upper/lower limit or a finite interval.

In the following subsections we will see the different PDFs used for each kind of rich value. As for the exact algorithm of detecting the type of rich value that corresponds to the final distribution, it is not explained but it can be inspected in the source code of the function `evaluate_distr` (check the `__init__.py` file inside the `richvalues` folder of the GitHub repository of the library). The usage of that function is explained in section 9.3.

8.1. Centered values

Let's consider a group of n variables x, with central values μ and uncertainties σ. This means that the probability density function (PDF) of the variable x is centered around μ with a width of the order of σ, so that the $1\,\sigma$ confidence interval (which includes 68.27 % of the distribution) is $(\mu - \sigma,\ \mu + \sigma)$.[27] We define the left and right amplitudes, a_1 and a_2, as the distances between the limits of the domain $(b_1,\ b_2)$ and the center, that is, $a_j = |b_j - \mu|$ for $j = 1,2$. Now, as these amplitudes can be different, we will split our desired PDF in two halves, one for $x < \mu$ and other for $x \geq \mu$, applying a little interpolation between them around the

26. Actually, in this simple case we can derive an analytical solution for the parameters, but that will not be the case in general.

27. There can be other interpretations, like $(\mu - \sigma,\ \mu + \sigma)$ being the *credibility* inte\rval, but we prefer to base our PDFs in quantiles (the median and the $1\,\sigma$ confidence interval).

median in order to have a smooth transition.

To propagate the uncertainties through a function f applied to the variables $\{x_i\}$, we can draw a sample of a large number of values (several thousands) of each variable x_i, apply the function to each of the elements of the samples, and then obtain a central value and an uncertainty for the resulting distribution. To do so, we need two things: an appropiate PDF for converting each variable x_i to a distribution of values, and a proper method to obtain a central value and an uncertainty from the resulting distribution. For the last task, we can use the mean, the median or the mode as the central value,[28] and the 1 σ confidence interval (68.27 %) to obtain the lower and upper uncertainty (with respect to the central value). As for the PDF, it will depend of the domain of the variable.

We will define our PDFs for the case of a variable x with a central value μ and an uncertainty σ, building the final PDF with two halves with amplitudes a_1 and a_2. Then, let's consider an amplitude a, which must be greater than the uncertainty, $a > \sigma$. In case we had lower and upper uncertainties, σ_1 and σ_2, we should just replace σ by σ_1 for the left half of the PDF and by σ_2 for the right half. In order to have a smooth transition between the two halves, a little interpolation between the two PDFs is done using a cosine function, preserving the median and the 1 σ confidence intervals. There is also an optional alternative for cases in which the amplitudes are large enough and the asymmetry of the uncertainty is low or medium.

In any case, any function f that we propose has to fulfill the following relations with μ, (σ_1, σ_2) and (a_1, a_2) defined by the quantiles:

$$\left.\begin{aligned} \int_{\mu-a_1}^{\mu+a_2} f(x)\,\mathrm{d}x &= 1 \\ \int_{\mu-a_1}^{\mu} f(x)\,\mathrm{d}x &= ½ \\ \int_{\mu-a_1}^{\mu-\sigma_1} f(x)\,\mathrm{d}x &= 0.15865\ldots \\ \int_{\mu-a_1}^{\mu+\sigma_2} f(x)\,\mathrm{d}x &= 0.84135\ldots \end{aligned}\right\} \quad . \tag{1}$$

This implies that the relation for the 1 σ confidence interval is satisfied: $\int_{\mu-\sigma_1}^{\mu+\sigma_2} f(x)\mathrm{d}x = 0.6827\ldots$

8.1.1. Normal distribution

If the domain of the variable x is $(-\infty, \infty)$, a proper function is the well-known Gaussian function:

$$f(x) = \frac{1}{\tau^{1/2}\sigma} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \quad , \tag{2}$$

with $\tau \equiv 2\pi$.[29] This would led to a normal distribution.

28. We prefer to use the median, as it is more robust to outliers and, specially, because it transforms directly when applying a function to a distribution of values; that is, the median of the resulting distribution of applying the function is just the function applied to the median of the original distribution.
29. See https://tauday.com.

8.1.2. Bounded normal distribution

If the domain of the variable is not $(-\infty,\ \infty)$, the normal distribution would be incorrect. Therefore, we have to use another function as the PDF.

Let's suppose a domain $(b_1,\ b_2)$. If the amplitude is quite greater than the uncertainty, $a \gg \sigma$, a good PDF would be just the normal distribution truncated to the domain $(b_1,\ b_2)$. However, for amplitudes closer to the uncertainty, it would be clearly incorrect, as the truncation modifies the confidence intervals, and thus the uncertainties.

To fix this, we make a variable change using the inverse of the hyperbolic tangent:

$$\frac{x-\mu}{a} \ \to\ \frac{\widetilde{x}-\mu}{a} \ \equiv\ \operatorname{arctanh}\left(\frac{x-\mu}{a}\right)\ . \tag{3}$$

When the argument of the inverse hyperbolic tangent is small, this function is like the identity (a line with slope 1), but when the argument approaches the values ± 1, it increases in absolute value, having vertical asymptotes in ± 1. Therefore, using this new variable $\widetilde{x}$ with a normal distribution, we are able to compress the original domain of the Gaussian of $(-\infty,\ \infty)$ to $(-a,\ a)$, as $|\widetilde{x}-\mu| \to \infty$ when $|x-\mu| \to a$. However, we have to make an additional modification to the PDF.

We want $\widetilde{x}$ to be a normal random variable. Thus, its PDF must be:

$$\widetilde{f}(\widetilde{x}) \,=\, \frac{1}{\tau^{1/2}\widetilde{\sigma}}\,\exp\left(-\frac{1}{2}\left(\frac{\widetilde{x}-\widetilde{\mu}}{\widetilde{\sigma}}\right)^2\right)\ , \tag{4}$$

where $\widetilde{\mu}$ and $\widetilde{\sigma}$ are the median and the standard deviation of the variable $\widetilde{x}$. We have a relationship between $\widetilde{x}$ and x (equation 2 in our case), that is, $\widetilde{x} = \widetilde{x}(x)$. This relation allows to express $\widetilde{\mu}$ and $\widetilde{\sigma}$ with respect to μ and σ: $\widetilde{\mu} = \widetilde{x}(\mu)$ and $\widetilde{\sigma} = \widetilde{x}(\sigma) - \mu$. In our particular variable change (equation 2), we have that $\widetilde{\mu} = \mu$ and $\widetilde{\sigma} = a\operatorname{arctanh}(\sigma/a)$. Now, our aim is to obtain the formula of the PDF with respect to x, showing also μ and σ.

The PDF is the probability density function, $f(x)$. This means that the probability of the variable to have a value between x and $\mathrm{d}x$ is $f(x)\,\mathrm{d}x$. We want this probability to be the same as that of a normal distribution with the variable $\widetilde{x}$, so it must be the same as $\widetilde{f}(\widetilde{x})\mathrm{d}\widetilde{x}$. As $\widetilde{x}$ can be expressed in terms of x, we have that $\mathrm{d}\widetilde{x} = (\mathrm{d}\widetilde{x}/\mathrm{d}x)\,\mathrm{d}x$, where $\mathrm{d}\widetilde{x}/\mathrm{d}x$ is the derivative of $\widetilde{x}$ with respect to x. Hence, the expression of our desired PDF would be:

$$f(x) \,=\, \frac{\mathrm{d}\,\widetilde{x}}{\mathrm{d}\,x}\,\widetilde{f}(\widetilde{x})\ . \tag{5}$$

Applying this formula to our particular variable change (equation 2) we finally obtain our desired PDF, which we will call *bounded Gaussian function*:

$$f(x) \,=\, \frac{1}{\tau^{1/2}\,a\,\operatorname{arctanh}(\sigma/a)}\,\frac{\exp\left(-\dfrac{1}{2}\left(\dfrac{\operatorname{arctanh}\left(\frac{x-\mu}{a}\right)}{\operatorname{arctanh}(\sigma/a)}\right)^2\right)}{1-\left(\dfrac{x-\mu}{a}\right)^2}\ . \tag{6}$$

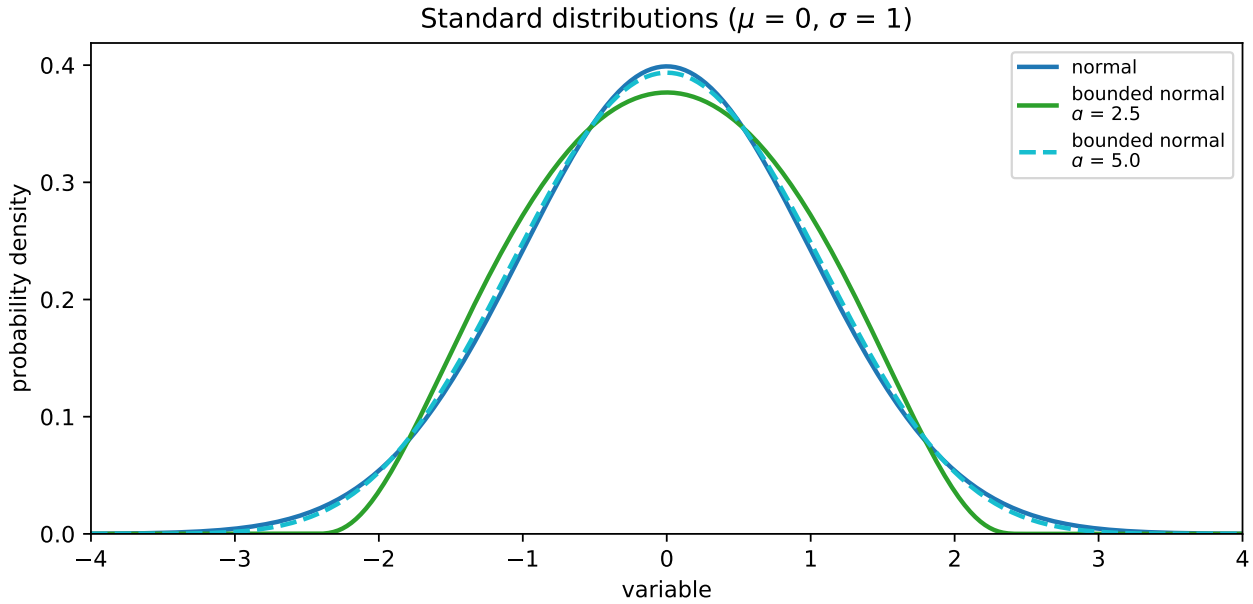

Figure 3. Probability density functions for a normal distribution and two bounded normal distributions.

This would led to a *bounded normal distribution*. As you can see in figure 3, it resembles a normal distribution but with a restricted domain. In fact, for small normalized uncertainties ($\sigma \ll a$) this PDF quickly tends to a Gaussian.

On the other hand, for big normalized uncertainties ($\sigma \sim a$), the shape of the PDF changes interestingly. For $\sigma/a \gtrsim 0.53$, the shape of the function starts to resemble that of a uniform distribution, but a change occurs at a value of $\sigma/a \simeq 0.61$. For normalized uncertainties greater than this limit value, two symmetric peaks appear in the PDF, forming a central dip or valley (see figure 4). This can seem unnatural, but it is the only way to achieve such high normalized uncertainties (σ/a).[30] As the uncertainty approaches the value of the amplitude, these peaks become higher and move closer to the domain edges.

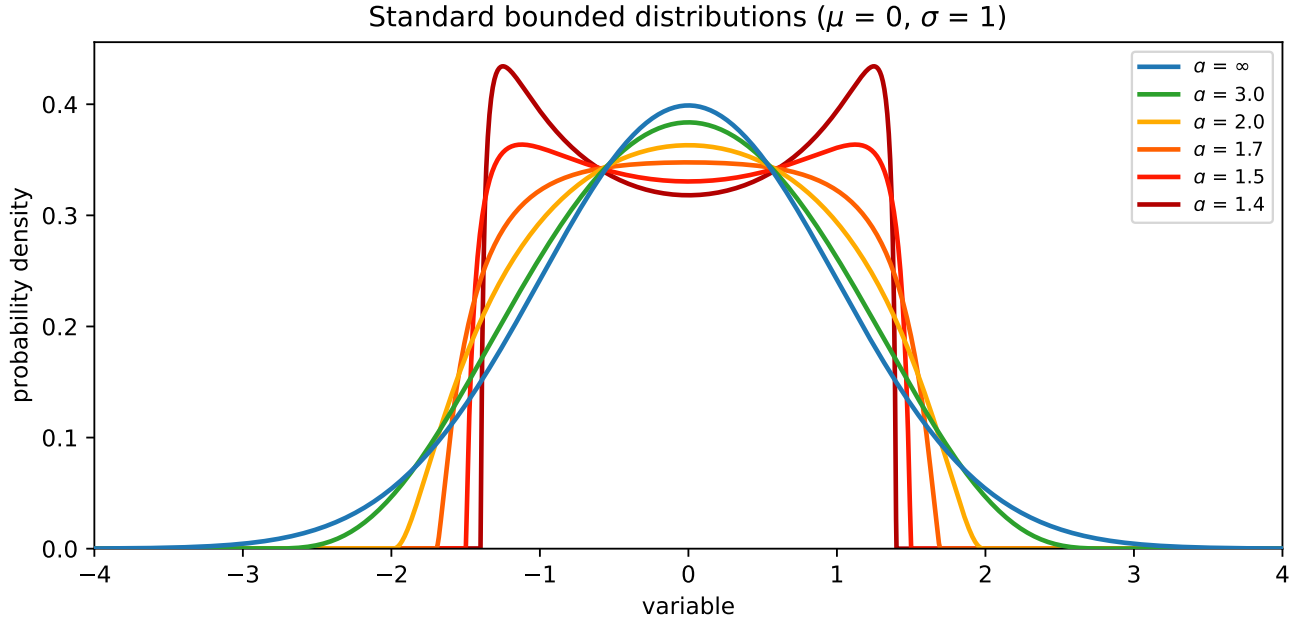

Figure 4. Probability density functions for bounded normal distributions with different amplitudes (a). The case of $a = \infty$ corresponds to a normal distribution.

8.1.3. Interpolation for asymmetric distributions

In many cases, centered rich values have a domain that is not symmetrical with respect to the central value, μ (for example, for positive variables). In other words, we would have two different amplitudes (a_1, a_2), one for the region below the central value ($x < \mu$) and one for the region over it ($x > \mu$). In such cases, we will build the probability density function (PDF) of the rich value joining the halves of two PDFs, one with each amplitude.

Similarly, if there is only one amplitude but we have asymmetrical uncertainties (lower and upper uncertainties), we would have to build the PDF joining the halves of two PDFs, one with each uncertainty (σ_1, σ_2).

In general, we can have one half of the PDF for $x < \mu$ with amplitude a_1 and uncertainty σ_1, and another half of the PDF for $x \geq \mu$ with amplitude a_2 and uncertainty σ_2. In most cases, there will be a gap in the union of the two functions (Δh), as the value at $x = \mu$ will be different for each half.

To avoid this discontinuity, which seems unnatural, we can add a correction to one of the PDF halves based on the cosine function. In particular, the corrected half will be the one with the lowest height at $x = \mu$. The domain of the correction will be $(\mu - \sigma_1, \mu)$ if these half is the one on the left side ($x < \mu$), and $(\mu, \mu + \sigma_2)$ if it is the one on the right side ($x \geq \mu$). Then, the correction itself will be a cosine function centered on μ and with a period of $4/3\ \sigma_i$ ($i = 1$ or 2), and raised to $3/2$. The amplitude of the correction will be equal to Δh for $|x - \mu| < ⅓\ \sigma_i$ but equal to $½\ \Delta h$ for $|x - \mu| > ⅓\ \sigma_i$.

In this way, the correction will be positive in the surroundings of the central part of the PDF ($|x - \mu| < ⅓\ \sigma_i$) and negative further away until reaching a distance from the center (μ) equal to the corresponding uncertainty (σ_i). Furthermore, the total area enclosed between the correction will be zero, so the 1 σ confidence interval will remain intact, and thus the uncertainties.

Figure 5 shows an example of the correction applied to a rich value which can only have positive values, that is, with a domain of $[0, \infty)$. As you can see, the correction allows to have a continuous and smooth transition between the two PDF halves.

In the rare case in which the size of the negative correction ($½\ \Delta h$) would make the PDF have a value too small in the region of the correction (in particular, ¼ of the minimum of the central heights of the two PDF halves), that size can be decreased: we can add an antisymmetrical correction (symmetric but negative) to the other half of the PDF; the sizes of the corrections on $x = \mu$ (let's call them c_1 and c_2) must follow that their sum is equal to the height difference of the PDF halves at $x = \mu$ (Δh), that is: $c_1 + c_2 = \Delta h$.

With this notation, we usually have that $c_2 = 0$ (the correction is done only on the half with lower height at $x = \mu$). If needed, we can reduce c_1 and increase c_2, with the extreme case of $c_1 = 0$ (the correction is done only on the half with higher height at $x = \mu$). The reason of using $c_2 = 0$ by default is that in this way the peak of the global PDF is μ, which seems a more natural choice than using $c_1 = 0$.

We call the resulting PDF *asymmetric bounded Gaussian* since it is made up of two

halves of bounded Gaussians (therefore, is asymmetrical) and then corrected with the interpolation around the median to be continuous.

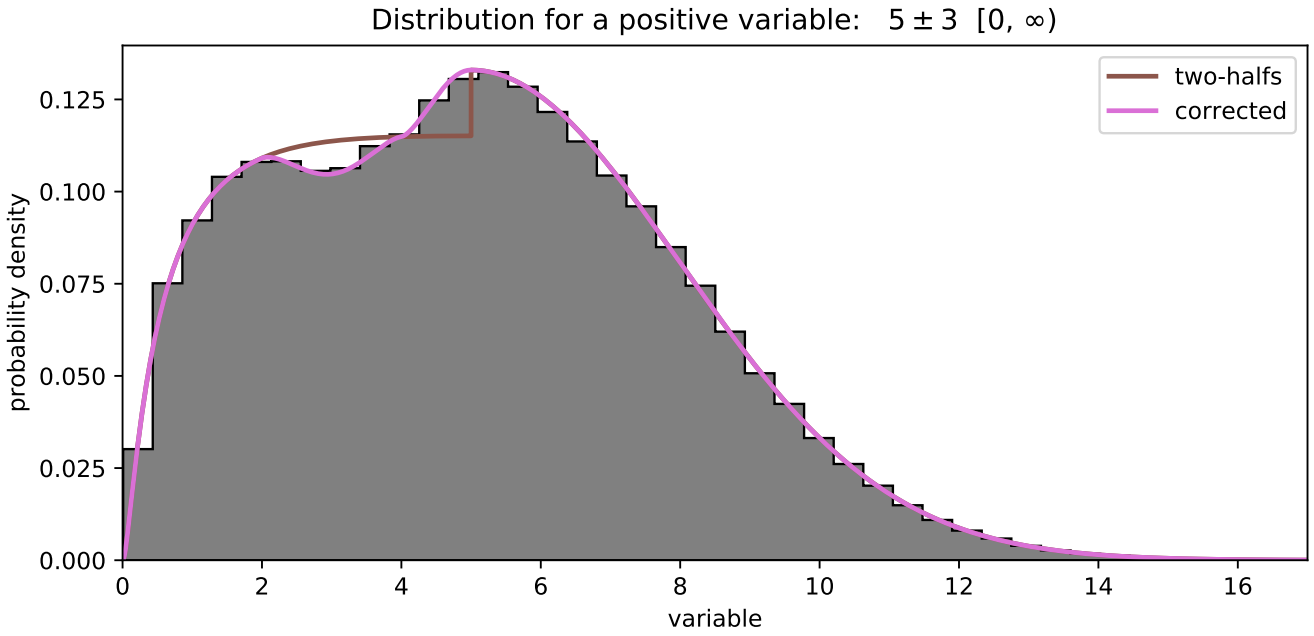

Figure 5. Probability density function corresponding to a rich value with positive domain and symmetric uncertainties. The brown curve is just the union of two PDF halves, one for each amplitude ($a = 5$ and $a = \infty$), while the pink curve has the correction using the cosine interpolation. Grey bars indicate the histogram of a sample drawn from the corrected PDF.

8.1.4. Alternative PDFs for asymmetric uncertainties

In some cases when the amplitudes are large enough and there is a low or medium asymmetry in the uncertainties, we can use alternative functions as the PDF of the rich value: the PDFs of a split normal distribution and a generative extreme value (GEV) distribution.[31]

In these cases with asymmetric uncertainties, the asymmetric bounded Gaussian corrected in the median can produce a bimodal curve, with a second peak due to the interpolation near the median to join the two halves of the bounded Gaussians. If this is the case, one can use these alternative PDFs. In any case, the asymmetric bounded Gaussian is already a function that perfectly meets our requirements with respect to (μ, σ_1, σ_2, a_1, a_2).

We will define the ratio between the greater uncertainty and the smaller one as $\chi_\sigma = \max(\sigma_2/\sigma_1,\ \sigma_1/\sigma_2)$, and we will also use the minimum value of the relative amplitude $\chi_a = \min(a_1/\sigma_1,\ a_2/\sigma_2)$ Depending on the value of these two values, we will be allowed to use alternative PDFs. However, this comes with an increase in computation time (specially for GEV functions), that is why by default the use of these alternative functions is disabled (this default behaviour can be changed; see Section 10 and the last two parameters). For the sake of completeness, we will explain both alternative functions, corresponding to the two mentioned distributions.

Split normal distribution

31. This implementation was inspired by the following work by Antonio Possolo et al. (2019): https://iopscience.iop.org/article/10.1088/1681-7575/ab2a8d/.

This distribution is formed by joining two halves of normal distributions with different standard deviations, scaled so that they match in the center. Therefore, its PDF can be described as the following:

$$f(x) \;=\; \begin{cases} \dfrac{2}{\tau^{1/2}\left(s_1+s_2\right)} \exp\left(-\dfrac{1}{2}\left(\dfrac{x-m}{s_1}\right)^2\right) , & x < m \\ \dfrac{2}{\tau^{1/2}\left(s_1+s_2\right)} \exp\left(-\dfrac{1}{2}\left(\dfrac{x-m}{s_2}\right)^2\right) , & x \geq m \end{cases} , \tag{7}$$

where here m is the mode (not the median) and (s_1, s_2) are the left and right widths, which correspond to the 1 σ credibility interval (not confidence) with respect to the mode. It happens that for low values of uncertainty asymmetry ($\chi_\sigma \lesssim 1.4$), there is a relation between (m, s_1, s_2) and (μ, σ_1, σ_2), defined by the three last conditions in equation 1. Actually, if $s_1 = s_2$, this function transforms to a Gaussian. For each set of values (μ, σ_1, σ_2), we can optimize the parameters (m, s_1, s_2) so that they fulfill the equations 1. In particular, a loss function is defined as the mean squared error (MSE) between the integrated mentioned quantiles and the expected values of ½, 0.15765… and 0.84135… Then, SciPy's function minimize, from its module optimize, is used to find the optimal parameter, using the Nelder-Mead method.

In RichValues, the criteria for using this kind of PDFs (if the use of split normal distributions is allowed) is that $\chi_a \geq 4$ and that $1 < \chi_\sigma \leq 1.3$. Then, if $1.3 < \chi_\sigma < 1.4$, we use a linear combination of a split-Gaussian and an asymmetric bounded Gaussian (or a GEV function, if this is allowed). Equivalently, for $3 < \chi_a < 4$, we also use a linear combination between the previously mentioned functions (depending on the value of χ_σ) and an asymmetric bounded Gaussian, so that for $\chi_a \leq 3$ we always use an asymmetric bounded Gaussian.

Generative extreme value (GEV) distribution

This distribution accepts three parameters: m, the mode; s, the scale (similar to a standard deviation); and ξ, the shape, that indicates the level of asymmetry of the curve (which in any case cannot be 0 even if $\xi = 0$. The exact PDF is as follows:

$$\frac{1}{s}\, t(x)^{\xi+1} \mathrm{e}^{-t(x)} , \qquad t(x) \;=\; \begin{cases} \left[1+\xi\left(\dfrac{x-m}{s}\right)\right]^{-1/\xi} , & \xi \neq 0 \\ \exp\left(-\dfrac{x-m}{s}\right) , & \xi = 0 \end{cases} . \tag{8}$$

It turns out that for low or medium values of uncertainty asymmetry ($\chi_\sigma \lesssim 2$), there is a relation between (m, s, ξ) and (μ, σ_1, σ_2), defined by the three last conditions in equation 1, as in the case for split normal distributions. Again, for each set of values (μ, σ_1, σ_2), we can optimize the parameters (m, s, ξ) so that they fulfill the equations 1. A loss function is defined as the mean squared error (MSE) between the integrated mentioned quantiles and the expected values of ½, 0.15765… and 0.84135… Then, SciPy's function minimize, from its module opti-

mize, is used to find the optimal parameter, using the Nelder-Mead method.

In RichValues, the criteria for using this kind of PDFs (if the use of split normal and GEV distributions is allowed) is that $\chi_a \geq 4$ and that $1.4 \leq \chi_\sigma \leq 2.0$. Then, if $1.3 < \chi_\sigma < 1.4$, we use a linear combination of a split-Gaussian and a GEV function. Also, if $2.0 < \chi_\sigma < 2.1$ we use a linear combination of a GEV function and an asymmetric bounded Gaussian. In this way, for $\chi_\sigma \geq 2.1$, we directly use an asymmetric bounded Gaussian. Lastly, if $3 < \chi_a < 4$, we use the a linear combination between the corresponding function mentioned before (depending on the value of χ_σ) and an asymmetric bounded gaussian, so that for $\chi_a \leq 3$ we just use an asymmetric bounded Gaussian; additionally, for the condition $1.4 \leq \chi_\sigma \leq 2.0$ we change the upper limit by 1.5 if $\chi_a = 3$ and an interpolation between 1.5 and 2.0 for $3 < \chi_a < 4$, so that the limit goes to 2.0 for $\chi_a = 4$.

8.2. Upper/lower limits and finite intervals

Lastly, we should also address the case of a variable with an upper/lower limit or even a finite interval. Let's consider an interval (x_1, x_2), which may be finite or infinite (and which can represent an upper/lower limit). If it is finite, we choose a uniform distribution between x_1 and x_2, with finite thresholds for 0, which we set to $\pm 10^{-90}$. But if it is infinite, we choose a symmetric loguniform distribution with finite thresholds for 0 and $\pm\infty$, which we set to $\pm 10^{-90}$ and $\pm 10^{90}$. For example, for an interval of $(-100, \infty)$, we would build a sample $\{x_-\}$ from a uniform distribution between –90 and 2 and a sample $\{x_+\}$ from a uniform distribution between –90 and 90. Our final distribution would be the joining of the samples of $\{-10^{\{x-\}}\}$ and $\{10^{\{x+\}}\}$.

8.3. Summary

If we have a set of variables x_i with central values μ_i and uncertainties σ_1, σ_2 we first build distributions $\{x_i\}$ using the mentioned PDFs. Then, we apply the function to the distributions, $f(\{x_i\})$, obtaining a new distribution. Finally, we use an algorithm[32] to detect if the distribution corresponds to an interval (that can be an upper/lower limit) or a defined value with uncertainties, and derive the corresponding parameters.

9. Additional useful functions

Besides the functions explained throughout this document, the `RichValues` libraries uses some additional functions to work. This section contains explanations of most of them, as they may be of interest.

9.1. Rounding numbers

The following functions are used for displaying the rich values with the correct number of sig-

32. The exact algorithm can be inspected in the source code of the function `evaluate_distr` (check the `__init__.py` file inside the `richvalues` folder of the GitHub repository: https://github.com/andresmegias/richvalues/). The usage of the `evaluate_distr` function is explained in section 9.3.

nificant figures, rounding the numbers accordingly to the type of rich value and, if existent, its uncertainties.

Function `round_sf`

If rounds the given number to the given number of significant figures. The arguments are:

- **`x`**. Input number.
- **`n`**. Number of significant figures. The default is `1`.
- **`min_exp`**. Minimum exponent to display the number in scientific notation. The default is `4`.
- **`lim_for_extra_sf`**. If the significand/mantissa of the number is lower than this limit value, the number will be displayed with an additional significant figure. By default it is `2.5`.

Function `round_sf_unc`

If rounds the given value and uncertainty to the given number of significant figures. The arguments are:

- **`x`**. Input value.
- **`dx`**. Uncertainty of the input value.
- **`n`**. Number of significant figures. The default is `1`.
- **`min_exp`**. Minimum exponent to display the numbers in scientific notation. The default is `4`.
- **`max_dec`**. Maximum number of decimals to be shown, to use the notation with parenthesis. The default is `5`.
- **`lim_for_extra_sf`**. If the significand/mantissa of the uncertainty is lower than this limit value, the numbers will be displayed with an additional significant figure. By default it is `2.5`.

Function `round_sf_uncs`

If rounds the given value and uncertainties to the given number of significant figures. The arguments are:

- **`x`**. Input value.
- **`dx`**. List containing the lower and upper uncertainties of the input value.

The rest of arguments are the same as in `round_sf_unc`: `n`, `min_exp`, `max_dec`, `lim_for_extra_sf`.

9.2. Creating distributions

The following functions are used for creating the distributions associated with the rich values.

Function `bounded_gaussian`

It applies the bounded Gaussian function defined in section 8.1.2, which is the probability density function (PDF) of a bounded normal distribution. The arguments are:

- `x`. Input array of values to apply the function.
- `m`. Median of the curve. By default it is `0`.
- `s`. Width of the curve (similar to the standard deviation). By default it is `1`.
- `a`. Amplitude of the curve (distance from the median to the domain edges). By default it is `np.inf`.

Function `asymmetric_bounded_gaussian`

It applies the asymmetric bounded Gaussian function defined in section 8.1.3, which is the probability density function (PDF) of a bounded normal distribution. The arguments are:

- `x`. Input array of values to apply the function.
- `m`. Median of the curve. By default it is `0`.
- `s1`. Left uncertainty (from the 1 σ confidence interval). By default it is `1.0`.
- `s2`. Right uncertainty (from the 1 σ confidence interval). By default it is `1.0`.
- `a1`. Left amplitude of the curve (distance from the median to the left domain edge). By default it is `np.inf`.
- `a2`. Right amplitude of the curve (distance from the median to the right domain edge). By default it is `np.inf`.
- `corrected`. Logical variable that determines if the correction of the interpolation around the median is applied, so that the final PDF is continuous. The default is `True`.

Function `splitgaussian`

It applies the split-Gaussian function defined in section 8.1.4, which is the probability density function (PDF) of a split normal distribution. The arguments are:

- `x`. Input array of values to apply the function.
- `m`. Mode of the curve. By default it is `0`.
- `s1`. Width of the left curve (similar to the standard deviation). By default it is `1.0`.
- `s2`. Width of the right curve (similar to the standard deviation). By default it is `1.0`.

Function `qsplitgaussian`

Split-Gaussian function, but defined with respect to quantile-related parameters. The arguments are:

- `x`. Input array of values to apply the function.
- `m`. Median of the curve. By default it is `0`.
- `s1`. Left uncertainty (from the 1 σ confidence interval). By default it is `1.0`.
- `s2`. Right uncertainty (from the 1 σ confidence interval). By default it is `1.0`.

Function `genextreme`

It applies the generative extreme value (GEV) function defined in section 8.1.4, which is the probability density function (PDF) of a GEV distribution. The arguments are:

- `x`. Input array of values to apply the function.

- **m**. Median of the curve. By default it is 0.
- **s**. Width of the left curve (similar to the standard deviation). By default it is 1.0.
- **e**. Shape parameter of the curve, that defines its asymmetry. By default it is 0.

Function qgenextreme

Generative extreme value (GEV) function, but defined with respect to quantile-related parameters. The arguments are:

- **x**. Input array of values to apply the function.
- **m**. Median of the curve. By default it is 0.
- **s1**. Left uncertainty (from the 1 σ confidence interval). By default it is 1.0.
- **s2**. Right uncertainty (from the 1 σ confidence interval). By default it is 1.0.

Function general_pdf

It applies the general PDF function for rich values as a combination of the asymmetrical bounded Gaussian, the split-Gaussian and the generative extreme value (GEV) function. The arguments are:

- **x**. Input values of the independent variable.
- **loc**. Median of the curve. By default it is 0.
- **scale**. Uncertainties (or uncertainty) from the 1 σ confidence interval. By default it is [1.0., 1.0].
- **bounds**. Boundaries of the independent variable (domain). By default it is [-np.inf, np.inf].
- **use_splitnorm**. Logical variable that determines if the split-Guassian is used to model the PDF when there is a low asymmetry in the uncertainties, when the amplitudes are large enough. By default it is False, since this increases the computation time.
- **use_genextreme**. Logical variable that determines if the GEV function is used to model the PDF when there is low/moderate asymmetry in the uncertainties, when the amplitudes are large enough. By default it is False, since this increases the computation time.

Function sample_from_pdf

It draws a sample from the distribution specified with the given probability density function (PDF). The arguments are:

- **pdf**. Input PDF of the distribution.
- **size**. Size of the sample.
- **low**. Minimum of the input values for the PDF.
- **high**. Maximum of the input values for the PDF.

Additionally, you can use any of the keyword arguments of the input PDF.[33] The values of

33. Keyword arguments are arguments that have a default value. For example, in bounded_gaussian and symmetric_loggaussian, the arguments m, s, and a are keyword arguments.

the arguments low and high should indicate the region where the input PDF is significantly greater than zero.

Function loguniform_distribution

It draws a sample from a lognormal distribution, with finite thresholds for 0 and $\pm\infty$. Arguments:

- **low**. Minimum of the input values for the PDF. By default it is -1.0.
- **high**. Maximum of the input values for the PDF. By default it is 1.0.
- **size**. Size of the sample. By default it is 1.
- **zero_log**. Decimal logarithm of the minimum value in absolute value that can be returned. By default it is -90.0.
- **infinity_log**. Decimal logarithm of the maximum absolute in absolute value that can be returned. By default it is 90.0.

Function distr_with_rich_values

It creates a distribution resulting from applying the given function to the given rich values, which will be represented by their corresponding distributions. It can also be called with the short name distribution. Arguments:

- **function**. Function to be applied to the input rich values.
- **args**. Input rich value arguments.
- **len_samples**. Size of the samples of the arguments. By default, it is the square root of the number of arguments times the default sample size (8 000).

The rest of the arguments are the same as in function_with_rich_values.

Function distr_with_rich_arrays

It creates a distribution resulting from applying the given function to the given rich arrays, whose elements will be represented by their corresponding distributions. It can also be called with the short name array_distribution. Arguments:

- **function**. Function to be applied to the input rich values.
- **args**. Input rich array arguments.
- **elementwise**. Logical variable that determines if the function is applied element by element to the input rich arrays, yielding another rich array with the same size as an output.

The rest of the arguments are the same as in function_with_rich_arrays.

9.3. Evaluating distributions

The following functions are used for evaluating distributions of numbers in order to interpret them as rich values. They can be used to interpret any random variable represented by a probability distribution as a rich value.

Function center_and_uncs

It returns the central value and uncertainties of the given distribution of numeric values. It can also be called as center_and_uncertainties. Arguments:

- **distr**. Input distribution of numbers.
- **center_function**. Function used to define the central value, possible values are: 'median', (by default), 'mean' and 'mode'.
- **interval**. Confidence interval, in percent, used to define the uncertainties, with respect to the central value. The default is the 1 σ confidence interval ($\sim$ 68.27 %), that is, 68.27. If center_function = 'mode', this would correspond to the credibility interval.
- **fraction**. Fraction of the input distribution (centered in the calculated central value) used to compute the uncertainties. By default, it is 1.0.

Function evaluate_distr

It interprets the given distribution as a rich value. It can also be called with the name evaluate_distribution. Arguments:

- **distr**. Input distribution of numbers.
- **domain**. Domain of the result, in case it is already known. By default it is ($-\infty$, ∞).
- **zero_log**. Decimal logarithm of the threshold in absolute value for ± 0, used to calculate the resulting domain and the range of the resulting distribution. By default it is -90.0.
- **infinity_log**. Decimal logarithm of the threshold in absolute value for $\pm\infty$, used to calculate the resulting domain and the range of the resulting distribution. By default it is 90.0.

Optionally, you can specify most of the keyword arguments of function_with_rich_values: function, args, len_samples, is_vectorizable, consider_vintervs, lims_fraction, and num_reps_lims. If so, the estimation of the rich value could improve a bit.

This function is used to represent variables defined by probability distributions different than the ones that are used in this library (section 8) as rich values. For example, consider a positive random variable defined by an exponential distribution with scale parameter 1, that is, with a probability density function (PDF) of $f(x) = e^{-x}$. We can use NumPy to define the PDF and create a rich value from it.

```
import numpy as np
rv.rval(lambda x: np.exp(-x), domain=[0,np.inf])
[out]   *0.7-0.5+1.1
```

The asterisk (*) is just indicating that this rich value has a custom PDF instead of the default one (section 8). The information of the PDF will be stored and used when performing operations with this rich value. We could obtain the same using the function sample_from_pdf (section 9.2).

```
distr = rv.sample_from_pdf(lambda x: np.exp(-x), size=int(1e4))
rv.evaluate_distr(distr, domain=[0,np.inf], consider_intervs=False,
                  save_pdf=True)
[out]   *0.7-0.5+1.1
```

Here, the argument save_pdf could have been omitted, although the information of the PDF would have been lost. Similarly, we could create the distribution first with NumPy and then evaluate it.

```
distr = np.random.exponential(scale=1, size=int(1e4))
rv.evaluate_distr(distr, domain=[0,np.inf], consider_intervs=False,
                  save_pdf=True)
[out]   *0.7-0.5+1.1
```

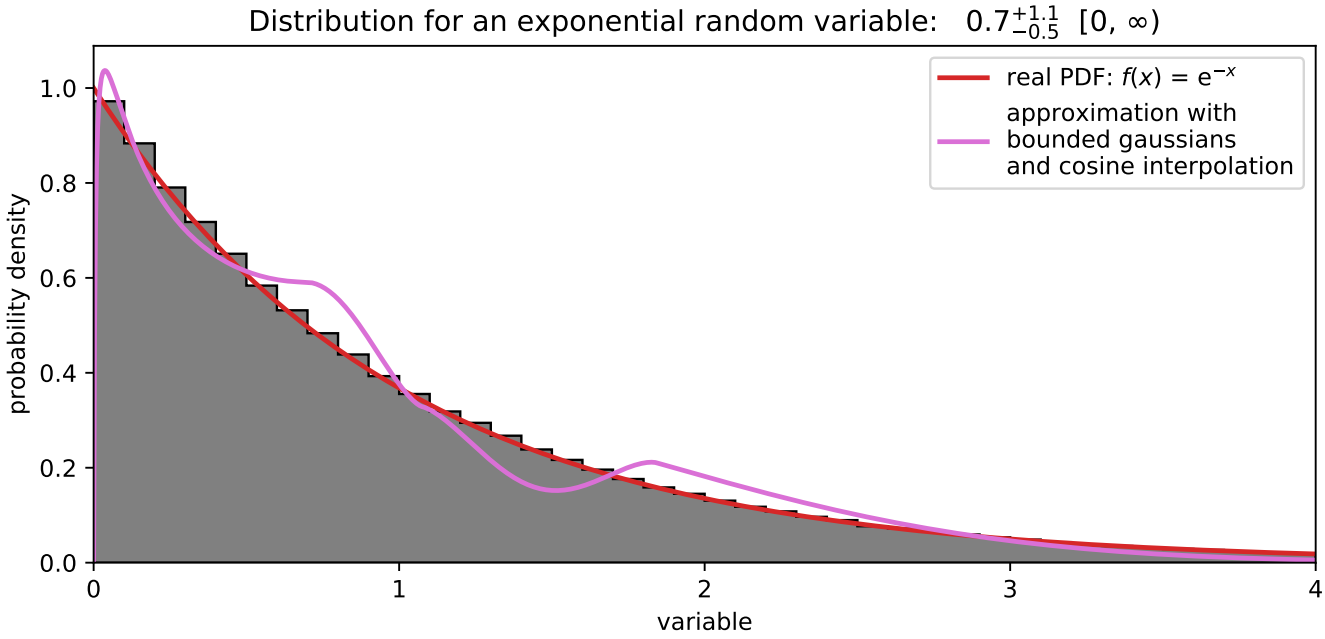

Figure 6. Histogram and probability density function corresponding to an exponential random variable with scale factor 1. The red curve is the real probability density function, whereas the pink curve is the approximated PDF resulting after interpreting the original distribution as a rich value. Grey bars indicate the histogram of a sample drawn from the real PDF.

You can see in figure 6 that interpreting the original exponential distribution as a rich value is a decent approximation, thanks to the functions explained in section 8.1.

Note that as rich values can be directly created specifying PDFs and then their information can be preserved and saved even for new rich values resulting from operations, this library can be used just with PDFs instead of using the numbers that define rich values (main value, uncertainties, upper/lower limit, etc.).

9.4. Comparing rich values

The following functions are used for comparing two rich values and also for comparing the elements of two rich arrays element-wise. They are used when defining the comparison special methods for the RichValue class (see section 4.1.4), These functions rely of the interval method of the RichValue class, that returns a confidence interval with a specified width (controlled by the sigmas argument) if it is a centered value, or the interval of possible values if it is an upper/lower limit or a finite range of values.

All of these functions have two mandatory arguments which are the two rich values/arrays that will be compared (x, y) and one or two optional arguments which correspond to the

sigmas argument of the interval method.

Function greater

It checks if the first input rich value (x) is greater than the second one (y), using the given sigmas as argument of the interval method of the RichValue class. By default, sigmas is equal to the default parameter sigmas for interval.

Function less

It checks if the first input rich value (x) is lower than the second one (y), using the given sigmas as argument of the interval method of the RichValue class. By default, sigmas is equal to the default parameter sigmas for interval.

Function equiv

It checks if the inputs rich values (x, y) are equivalent, using the given sigmas as argument of the interval method of the RichValue class. By default, sigmas is equal to the default parameter sigmas for overlap.

Function greater_equiv

It checks if the first input rich value (x) is greater or equivalent to the second one (y), using the above functions greater and equiv. The optional argument sigmas_interval is used as the sigmas argument of the greater function, and the other optional argument sigmas_overlap is used as the sigmas argument of the equiv function. By default, sigmas_interval is equal to the default parameter sigmas for interval and sigmas_overlap is equal to the default parameter sigmas for overlap.

Function less_equiv

It checks if the first input rich value (x) is less or equivalent to the second one (y), using the above functions less and equiv. The optional argument sigmas_interval is used as the sigmas argument of the less function, and the other optional argument sigmas_overlap is used as the sigmas argument of the equiv function. By default, sigmas_interval is equal to the default parameter sigmas for interval and sigmas_overlap is equal to the default parameter sigmas for overlap.

9.5. Smoothing curves

In order to estimate the PDF shape corresponding to a distribution of values, they are grouped into bins (like in a histogram). Then, to smooth the resulting curve, a smoothing is applied in the form of a rolling average with size equal to 5. To apply that, a function called rolling_function is used.

Function rolling_function

It applies a function in a rolling way to the given data, in windows of the specified size. The arguments:

- **func**. Function to be applied.

- **x**. Input data to apply the function.
- **size**. Size of the windows to group the data. It should be odd and greater than 1.

Apart from these arguments, we can include more keyword arguments corresponding to the input function. This function uses another internal function (rolling_window) written by Erik Rigtorp.

9.6. Implemented NumPy functions

The are several NumPy functions that have been implemented into the RichValues library with the same name, so that they can operate with rich values and/or rich arrays. Below is a list with the names of these functions; you can check their features and their usage on NumPy's documentation.[34]

- Functions for masking arrays: isnan, isinf, isfinite.
- Functions for concatenating arrays: append, concatenate.
- Mathematical functions: mean, sqrt, exp, log, log10, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, arctanh.

Additionally, there are several NumPy functions that do work with rich values/arrays inputs theirselves; for example, np.max, np.argmax, np.minimum, np.sort and np.dot.

10. Changing the default parameters

Most of the default parameters of the functions and classes of this library can be modified through the variable defaultparams. This is a Python dictionary containing the values of the default variables with a specific name. Below is a list with all the parameter names, the corresponding variable names used in the library and their description:

- **domain**. Domain of the rich values, that is, the minimum and maximum values that the variables associated with the rich values can take. By default it is all the real numbers, that is, [-np.inf, np.inf].
- **size of samples**. Size of the sample of the distribution associated with the rich value used to estimate the uncertainty propagation (usually len_samples). By default it is 8000.
- **number of significant figures**. Number of significant figures to display the rich values (usually num_sf).[35] By default it is 1.
- **minimum exponent for scientific notation**. Minimum exponent used to display the number in scientific notation (usually min_exp). By default it is 4.
- **maximum number of decimals to use parenthesis**. Maximum number of decimals to be displayed, to start using the notation with parenthesis (usually max_dec). By default it is 5.

34. https://numpy.org/doc .
35. If there is no uncertainty, the number will be showed with an additional significant figure.

- **limit for extra significant figure**. If the significand/mantissa[36] of the uncertainty of the rich value (if it is a central value with uncertainty) or its main value (if it is an upper/lower limit, or an edge of an interval) is lower than this limit number, the rich value will be displayed with an additional significant figure. By default, this value is `2.5`, and it is called `extra_sf_lim` in some functions; for example, a rich value with a central value of 5.21 and an uncertainty of 0.42 would be displayed as `5.2 +/- 0.4`, but if the uncertainty was 0.12, it would be `5.21 +/- 0.12`. If you want that limit to be excluded from the extra significant figure, you can define this variable as your limit minus a tiny number; for example: `2.5 - 1e-8`.
- **use extra significant figure for exact values**. Logical variable that determines if exact rich values (centered values with no uncertainty) are displayed with one extra significant figure. By default it is `True`.
- **use extra significant figure for finite intervals**. Logical variable that determines if the edges of constant ranges of values are displayed with one extra significant figure. By default it is `True`.
- **omit ones in scientific notation in LaTeX**. Logical variable that allows to remove the ones in scientific notation when displaying exact rich values or upper/lower limits in LaTeX formatting style. By default it is `False`.
- **multiplication symbol for scientific notation in LaTeX**. Symbol to be used when displaying a value in scientific notation in LaTeX mathematical mode for the multiplication of the significand/mantissa and the decimal power. By default it is `\\cdot`.[37]
- **sigmas to define upper/lower limits from read values**. If a rich value with a central value and uncertainties has one bound of its $1\,\sigma$ interval which surpasses one one of the domain edges, the rich value will be reconsidered as an upper/lower limit. Then, the main value will be the original central value minus/plus this parameter times the lower/upper uncertainty.
- **sigmas to use approximate uncertainty propagation**. This value defines when to apply the analytic approximation for uncertainty propagation in the operations with rich values. If the minimum value of the set of the signal-to-noise ratios plus the relative amplitudes is greater than this value, the approximation will be applied instead of creating samples from the distributions associated with the rich values, hence reducing the computation time. By default it is `20.0`, and it is called `sigmas` in some functions.
- **use 1-sigma combinations to approximate uncertainty propagation**. Logical variable (usually called `use_sigma_combs`) that determines if the propagation of uncertainties is approximated with the use of the corresponding function applied to the central value plus and minus its uncertainties. By default it is `False`.
- **fraction of the central value for upper/lower limits**. Variable used in `function_with_rich_values` with the name `lims_fraction`. In case the rich value resulting of applying

36. The number multiplying the decimal power.

37. In Python, two backslashes (\\) are needed in order to display just one (\) in a text string.

the corresponding function is an upper/lower limit, this factor is used to calculate the limit. It can take values from 0 to 1, and the closer it is to 1, the closer the resulting limit will be to the function applied to the central/limit value of the arguments. By default it is 0.2.

- **number of repetitions to estimate upper/lower limits**. Variable used in function_with_rich_values with the name num_reps. It is the number of repetitions of the sampling done in the cases of having an upper/lower limit for better estimating its value. By default it is 4. Greater values are recommended if fraction of the central value for upper/lower limits is lower than 0.2.
- **decimal exponent to define zero**. When performing operations and creating samples from distributions, any number lower in absolute value than the decimal power of this value will be considered as 0. By default it is -90.0.
- **decimal exponent to define infinity**. When performing operations and creating samples from distributions, any number lower in absolute value than the decimal power of this value will be considered as ∞. By default it is 90.0.
- **sigmas for interval**. Size of the confidence interval in units of the uncertainties used to compare two rich values with comparison operators (<, >, <= >=) and determine which of then is lower and which is greater. It corresponds to the argument sigmas of the interval method of the RichValue class. By default it is 3.0.
- **sigmas for overlap**. Size of the confidence interval in units of the uncertainties used to determine if two rich values are equivalent when using the equal operator (==), in the sense that their confidence intervals overlap. It corresponds to the argument sigmas of the interval method of the RichValue class. By default it is 1.0.
- **assume integers**. Logical variable that determines if new rich values are of integer nature. By default it is False.
- **save PDF in rich values**. Logical variable that determines if new rich values will store explicitly the probability density function (PDF) associated with each of them.
- **show domain**. Logical variable that determines if the domain of a rich value is shown when it is displayed. By default it is False.
- **show asterisk for rich values with custom PDF**. Logical variable that determines if an asterisk (*) is shown when a rich value with a custom probability density function (PDF) is displayed. By default it is True.
- **use split normal distributions to model rich values**. If this logical variable is set to True, the PDF of a split normal distribution will be used to represent rich values with slightly asymmetrical uncertainties. This icnreases increasing the computation time.
- **use generative extreme value distributions to model rich values**. If this logical variable is set to True, the PDF of a generative extreme value (GEV) distribution will be used to represent rich values with moderate asymmetrical uncertainties. This increases the computation time. The previous default parameter has to be to True (use split normal distributions) to take this one into consideration.

In order to change any of the default parameters, you can use the function set_default_params, using a dictionary containing the parameter names and their new values as an input. For example:

```
rv.set_default_params({'limit for extra significant figure': 2.0})   .
```

To restore the original default parameters, you can call the function restore_default_params, which does not have any arguments.

11. Short and full names for functions and more

Some of the functions of the RichValues library can be called with two names: the full name and an abbreviated one. For example, the function rich_value can also be called just rval. Similarly, most of the arguments of the functions and classes that create rich values have short and full names, and the same goes for instance variables and methods for the different classes of the library. Below is a table with all the relations of short and full names, with the longer names on the left and the shorter names on the right.

Function names	
rich_value	rval
rich_array	rarray
rich_dataframe	rdataframe, rich_df
function_with_rich_values	function
function_with_rich_arrays	array_function
distr_with_rich_values	distribution
distr_with_rich_arrays	array_distribution
evaluate_distribution	evaluate_distr
center_and_uncertainties	center_and_uncs

RichValue class instance variables	
number_of_significant_figures	num_sf
maximum_number_of_decimals	max_dec
minimum_exponent_for_scientific_notation	min_exp
limit_for_extra_significant_figure	extra_sf_lim

RichValue class attributes / methods	
is_limit	is_lim
is_interval	is_interv
is_centered	is_centr
is_constant	is_const
is_infinite	is_inf
is_not_a_number	is_nan
relative_uncertainty	rel_unc
signal_to_noise	signal_noise
amplitude	ampl
relative_amplitude	rel_ampl

RichValue class arguments / instance variables	
main_value	main
uncertainty	unc
is_lower_limit	is_lolim
is_upper_limit	is_uplim
is_finite_range	is_range
is_integer	is_int

ComplexRichValue class arguments / instance variables	
real_part	real
imaginary_part	imaginary, imag

ComplexRichValue attributes	
module	mod
angle	ang

RichArray class arguments / attributes	
main_values	mains
uncertainties	uncs
are_lower_limits	are_lolims
are_upper_limits	are_uplims
are_finite_ranges	are_ranges
are_integers	are_ints

RichArray class attributes / methods	
numbers_for_significant_figures	nums_sf
minimum_exponents_for_scientific_notation	min_exps
maximum_numbers_of_decimals	max_decs
limits_for_extra_significant_figure	extra_sf_lims
are_limits	are_lims

normalized_uncertainty	norm_unc	are_intervals	are_intervs
propagation_score	prop_score	are_centered	are_centrs
variance	var	are_constants	are_consts
standard_deviation	std, stdev	are_infinites	are_infs
probability_density_functio n	pdf	are_not_a_numbers	are_nans
set_limits_uncertainty	set_lims_unc	relative_uncertainties	rel_uncs
RichDataFrame class methods		signals_to_noises	signals_noises
get_parameters	get_params	amplitudes	ampls
set_parameters	set_params	relative_amplitudes	rel_ampls
set_limits_uncertainties	set_lims_uncs	normalized_uncertainties	norm_uncs
function_with_rich_values function arguments		propagation_scores	prop_scores
arguments	args	standard deviations	stds, stdevs
samples_length, samples_size	len_samples	set_parameters	set_params
samples_length, samples_size	len_samples	set_limits_uncertainties	
uncertainty_function	unc_function	~	

The extended names for the arguments of function_with_rich_values that appear on the table are also valid for the functions function_with_rich_arrays, distr_with_rich_values, and evaluate_distr. Note that the argument domain, present in most of the functions and classes mentioned in the table, has not any additional name; the same goes for the argument/method domains for the RichArray class.

Citation of the library

If you use RichValues for your work, it would be great if you cite it. You can put the link to the GitHub repository, where this user guide can also be downloaded:

https://github.com/andresmegias/richvalues/ .

12. Useful links

The following links may be of interest:

- **Python.**
  https://www.python.org/
- **NumPy.**
  https://numpy.org/
- **Pandas.**
  https://pandas.pydata.org/
- **SciPy.**
  https://scipy.org/
- **Matplotlib.**
  https://matplotlib.org/

Credits

This software has been developed at the Centre for Astrobiology (*Centro de Astrobiología*, CAB), in Madrid (Spain), within the group of Chemical Complexity in the Interstellar Medium and Star Formation (Department of Astrophysics).

Coding and testing
Andrés Megías Toledano

Testing
Álvaro López Gallifa
David San Andrés de Pedro

Discussion
Izaskun Jiménez Serra
Jacobo Aguirre Araujo
Miguel Á. Cerviño Saavedra
Marcos Jiménez Martínez
Eva Herrero Cisneros

License

The library RichValues is published under a BSD 3-Clause "New" or "Revised" License.