PyUncertainNumber

(Leslie) Yu Chen & Ioanna Ioannou & Scott Ferson

Jan 18, 2025

CONTENTS

bla bla …

CHAPTER 1

UNCERTAINTY CHARACTERISATION

1.1 variability and incertitude

Partial specification given information, bla bla ..

Tip

It is suggested to use interval analysis for propagating ignorance and the methods of probability theory for propagating variability.

See also

See also in the *propagation*

1.2 bounding distributional parameters

The mean of a normal distribution may be elicited from an expert but this expert cannot be precise to a certain value but rather give a range based on past experience.

To comprehensively characterise a pbox, specify the bounds for the parameters along with many other ancillary fields.

```
from pyuncertainnumber import UncertainNumber as UN

e = UN(
    name='elas_modulus',
    symbol='E',
    units='Pa',
    essence='pbox',
    distribution_parameters=['gaussian', ([0,12],[1,4])])
```

In cases where one wants to do computations quickly.

```
import pyuncertainnumber as pun
un = pun.norm([0,12],[1,4])
```

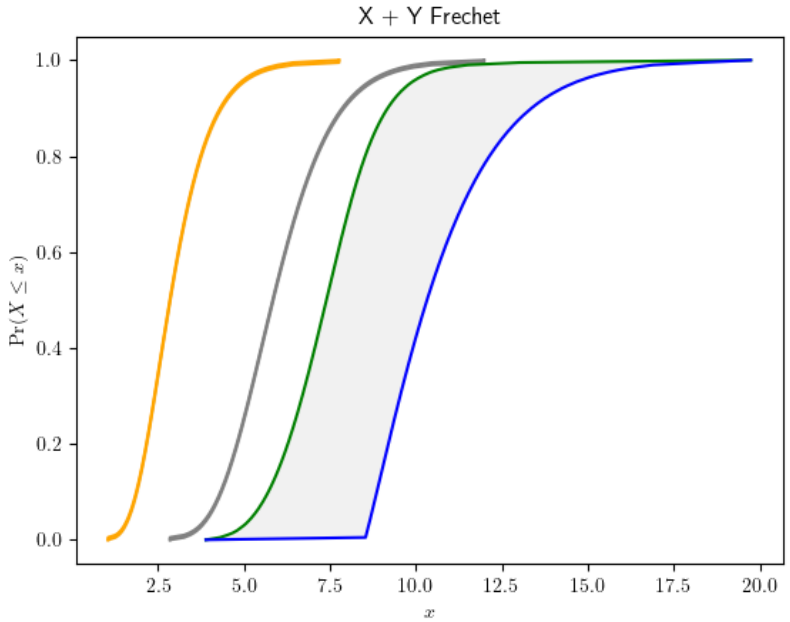
1.3 aggregation of multiple sources of information

Expert elicitation has been a challenging topic, especially when knowledge is limited and measurements are sparse. Multiple experts may not necessarily agree on the choice of elicited prbability distributions, which leads to the need for aggregation. Below shows two situations for illustration.

Assume the expert opinions are expressed in closed intervals. There may well be multiple such intervals from different experts and these collections of interval can be overlapping, partially contradictory or even completely contradictory. Their relative credibility may be expressed in probabilities. Essentially such information creates a **Dempster-Shafer structure**. On the basis of a mixture operation, such information can be aggregated into a **pbox**.

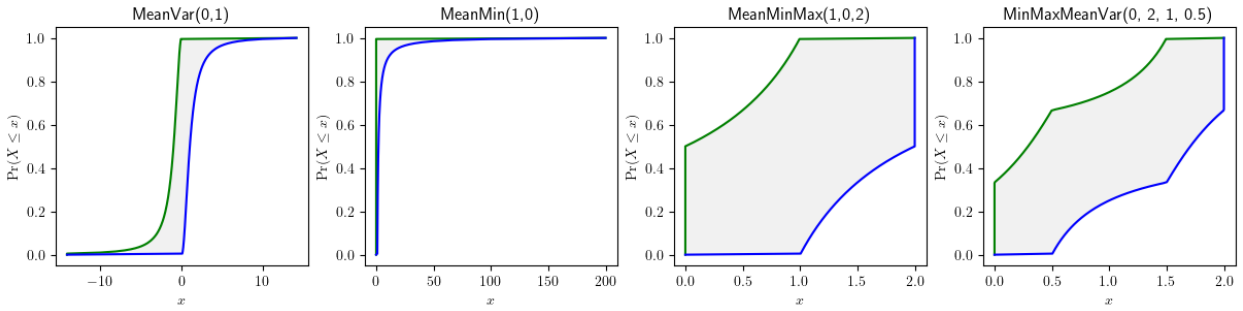
> **Tip**
>
> The different sub-types of uncertain number can normally convert to one another (though may not be one by one), ergo the uncertain number been said to be a unified representation.

Pbox arithmetic also extends the convolution of probability distributions which has typically been done with the independence assumption. However, often in engineering modelling practices independence is assumed for mathematical easiness rather than warranted. Fortunately, the uncertainty about the dependency between random variables can be characterised by the probability bounds, as seen below. It should be noted that such dependency bound does not imply independence.



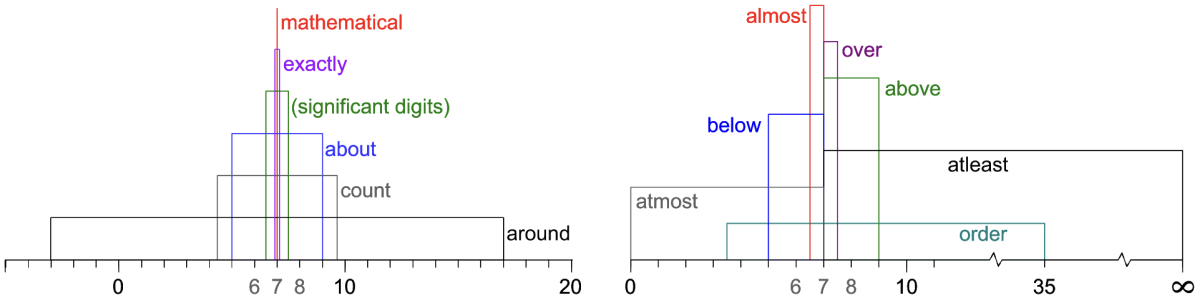
1.4 known statistical properties

When the knowledge of a quantity is limited to the point where only some statistical information is available, such as the *min*, *max*, *median* etc. but not about the distribution and parameters, such partial information can serve as **constraints** to bound the underlying distribution:



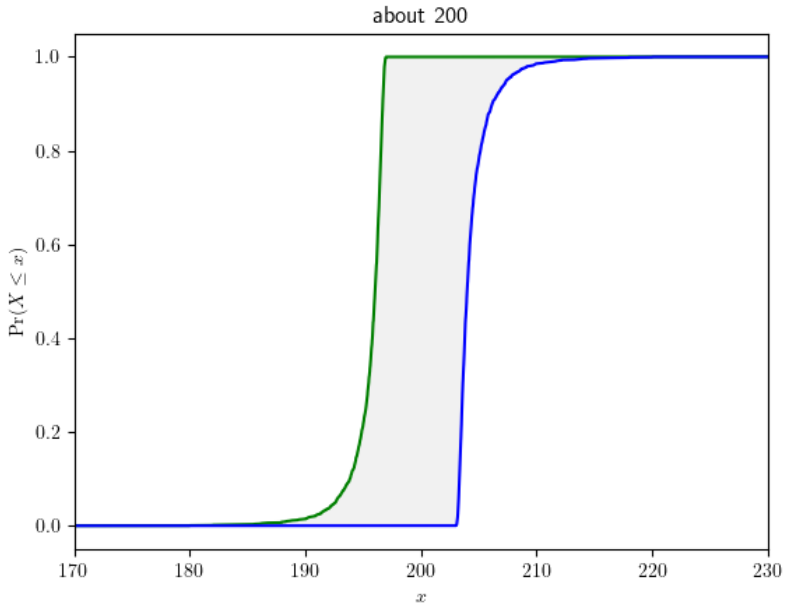
1.5 hedged numerical expression

Sometimes only purely qualitive information is available. An important part of processing elicited numerical inputs is an ability to quantitatively decode natural-language words, the linguistic information, that are commonly used to express or modify numerical values. Some example include 'about', 'around', 'almost', 'exactly', 'nearly', 'below', 'at least', 'order of', etc. A numerical expression with these approximators are called *hedges*. Extending upon the significant-digit convention, a series of interval interpretations of common hedged numerical expressions are proposed.



Besides intervals, `PyUncertainNumber` also supports interpreting hedged expressions into p-boxes. As an example, assume one wants to find out what "about" is about in terms of the uncertainty. The syntax and result is shown below:

```
import pyuncertainnumber as pun
pun.hedge_interpret('about 200', return_type='pbox').display()
```



1.6 interval measurements

CHAPTER 2

PROPAGATION

Methods to efficiently propagate different types of uncertainty through computational models are of vital interests. `PyUncertainNumber` includes strategies for black box uncertainty propagation as well as a library of functions for PBA (Probability Bounds Analysis) if the code can be modified. `PyUncertainNumber` provides a series of uncertainty propagation methods.

It is suggested to use *interval analysis* for propagating ignorance and the methods of probability theory for propagating variability. But realistic engineering problems or risk analyses will most likely involve a mixture of both types and as such probability bounds analysis provides means to rigourously propagate the uncertainty.

For aleatoric uncertainty, probability theory already provides some established approaches, such as Taylor expansion or sampling methods, etc. This guide will mostly focuses on the propagation of intervals due to the close relations with propagation of p-boxes.

2.1 Vertex method

The vertex propagation method (Dong and Shah, 1987) is a straightforward way to project intervals through the code, by projecting a number of input combinations given by the Cartesian product of the interval bounds. This results in a total of $n = 2d$ evaluations, where d is the number of interval-valued parameters. In the case of two intervals, $[x]$ and $[y]$, the code, $f()$ must be evaluated four times at all endpoints. The main advantage of the method is its simplicity. But it should be used under the assumption of monotonicity.

2.2 Subinterval reconstitution

To accommodate the presence of non-monotonic trends, the input intervals can be partitioned into smaller intervals, which can then be propagated through the model using vertex propagation and an output interval can be reassembled. The logic behind this method is that even though the code may not be monotonic over the full width of the interval, it will likely behave monotonically on a smaller interval, provided the underlying function is pathologically rough. Thus, output intervals for even highly non-linear functions can be computed to arbitrary reliability.

$$f(X) = \cup_{j=1}^{n} f(X_j) \subseteq \cup_{j=1}^{n} F(X_j)$$

where $\cup_{j=1}^{n} X_j = X$ and F denotes an interval extension of f.

2.3 Cauchy-deviate method

Similar to the Monte Carlo methods, the Cauchy-deviate method is also built upon the direct sampling of uncertain input numbers expressed as intervals. This makes it suitable for propagating epistemic uncertainty through black-box models. For each uncertain number, the method generates random samples from a Cauchy distribution and computes the width of the output interval through appropriate scaling of these samples. If the model has more than one input, the errors in each are considered independent at the sampling step. By generating samples outside of the input interval bounds, the Cauchy-deviate method explicitly includes those bounds via normalisation, thus solving one of the problems classical sampling methods face when used to propagate epistemic uncertainty.

2.4 Gradient-based optimisation methods

These methods start with a set of initial input values and by searching for new candidates which yield better solutions than the previous iterations find an optimum solution. To ensure that the algorithm converges as soon as possible, for each iteration a single best solution is identified. Key characteristic of these methods is that, for each iteration, the search for better solution is local, in other words it takes place in the immediate neighbourhood of the input values used in the previous iteration, leading to a local optimum solution. (Stork et al., 2022) compared the way these methods work with a mountaineer on a mission to find and climb to the summit of the tallest mountain in an area.

Method	Underlying Assumptions	Cost	Rigor*
Interval Arithmetic	Access to the code/ No repeated variables	Low	High
Vertex Propagation	Monotonic over intervals	2^d	High
Subinterval Reconstitution	Monotonic in subintervals	$(m+1)^d$	High
Monte Carlo methods	Intervals as uniform distributions	m	Low
Cauchy-Deviate Method	Linear over intervals	m	High
Gradient-based Optimisation	Unimodal Function	$< m$	High
Gradient-free Optimisation	No	$< m$	High

CHAPTER 3

PROBABILITY BOX

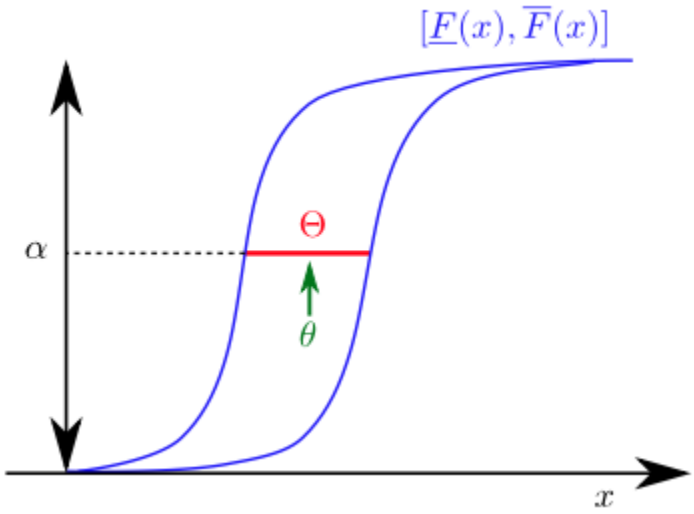
A conspicuous problem in probability distribution elicition, for example in probabilistic modelling analysis, is that the specification is typically precise, despite hardly justified by empirical informtion in many cases.

> **Attention**
>
> epistemic uncertainy remains on the shape, parameters, and dependencies of the distributions.

Probability box (abbreviated as `p-box`) essentially represents bounds on the cumulative distribution function (c.d.f) of the underlying random variable. Let $[\overline{F}, \underline{F}]$ denotes the **set** of all nondecreasing functions from the reals into $[0, 1]$ such that $\underline{F} \leq F \leq \overline{F}$. This means that, $[\overline{F}, \underline{F}]$ denotes a p-box for a random varaible X whose c.d.f F is unknown except that it is within the "box" circumscribed by the lower ($\underline{F}$) and upper bound ($\overline{F}$).

$$\underline{F} \leq F(x) \leq \overline{F}$$

p-box collectively reflects the variability (aleatoric uncertainty) and incertitude (epistemic uncertainty) in one structure for the uncertain quantity of interest. The horizontal span of the probabilty bounds are a function of the variability and the vertical breadth of the bounds is a function of ignorance.



Hint

There is a storng link between p-box and Dempster-Shafer structures (which `PyUncertainNumber` also explicitly provides support :boom:). Each can be converted to the other. However, it should be noted such translation is not one-to-one.

`PyUncertainNumber` provides support for operations with p-boxes ranging from *characterisation*, aggregation, propagation. Go check out these links for details as to the computation with p-boxes. Meanwhile, quick examples show below:

```
from PyUncertainNumber import UncertainNumber as UN

un = UN(
    name='elas_modulus',
    symbol='E',
    units='Pa',
    essence='pbox',
    distribution_parameters=['gaussian', [(0,12),(1,4)]])
_ = un.display(style='band')
```

CHAPTER 4

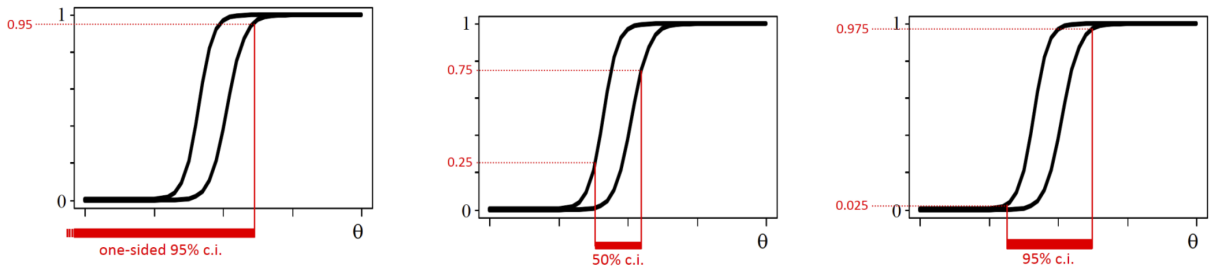
CONFIDENCE BOX

Traditionally confidence interval is widely used in engineering to make inference with respect to parameters of interest. It is appealing as it provides a gurantee of statistical performance through repeated use. Studies in frequetist inference continue to wonder the possibility of a *distributional estimator* just like the Bayesian posterior, which leads to significant developments towards the **confidence distribution**, though it is not technically the same as a distribution in the canonical sense.

> **Tip**
>
> A confidence distribution is essentially a ciphering device that encodes confidence intervals for each possible confidence level.

The confidence distribution conveniently provides confidence intervals of all levels for a parameter of interest. **Confidence boxes**, or confidence structures[1], (abbreviated as c-box) are imprecise generalisations of confidence distributions and they can be applied to problems with discrete observatons, interval-censored data, and even inference problems in which no assumption about the distribution shape can be made.

The figure below provides an illustration of the confidence box which yields several confidence intrevals for the parameter θ.



[1] Balch, Michael Scott. "Mathematical foundations for a theory of confidence structures." International Journal of Approximate Reasoning 53.7 (2012): 1003-1019.

An example of Gaussian distributed parameters

The cumulative confidence distribution for μ can be defined as:

$$H(\mu, \mathbf{x}) = F_{t_{n-1}}\left(\frac{\mu - \bar{x}}{s_x/\sqrt{n}}\right)$$

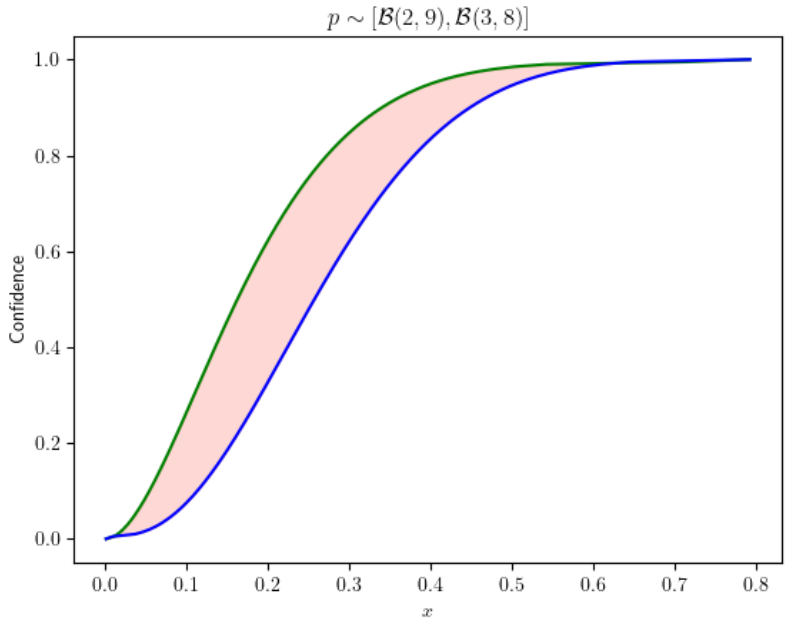
and also the distributional estimator for the parameter σ^2 is:

$$H_{\chi^2}(\sigma^2, \mathbf{x}) = 1 - F_{\chi^2_{n-1}}\left(\frac{(n-1)s_x^2}{\sigma^2}\right)$$

where $F_{t_{n-1}}$ is the c.d.f for a t distribution with $n-1$ degrees of freedom while $F_{\chi^2_{n-1}}$ is the c.d.f of the χ^2_{n-1} distribution

Generallly one cannot directly compute with confidence intervals, but you can compute with confidence boxes from which you can get arbitrary confidence intervals for the results. To facilitate the analysis, `PyUncertainNumber` provides straightforward syntax to derive the sample-dependent confidence box, as shown below:

```
# x[i] ~ binomial(N, p)
# k=2, n=10
c = infer_cbox('binomial', data=[2], N=10)
```



CHAPTER 5

INTERVAL ANALYSIS

Intervals play a central role in the probability bounds analysis and has been discussed in the context of computational functional analysis. It is known for making rigourous computations in terms of three kinds of errors in numerical analysis: rounding errors, truncation errors, and input errors.

5.1 Interval arithmetic

Hint

the key point of the definitions of basic arithmetic operations between intervals is **computing with intervals is computing with sets**.

Interval arithmetic operations can be defined as :

$$X \odot Y = \{x \odot y : x \in X, y \in Y\}$$

where $\odot$ stands for the elementary binary operations such as addition or product etc.

A key consideration of the propagation of interval objects is the *dependency* issue, which hinders the naive uses of interval arithmetic in many problems as it often yields inflated interval outputs. The *image set* under a real-valued function mapping f as x varies through a given interval $[X]$ (or simply X) can be defined as:

Note

Square backets are used to visually hint the nature of an Interval typed variable. In Python, square brackets suggest a list datatype which is ubiquitous, as such, in `PyUncertainNumber` we provide a parser for easy creation of interval objects with lists.

$$f(X) = \{f(x) : x \in X\}$$

It should be noted that when interval arithmetic is naively used in computations, it may not necessary yield the best-possible (or sharpest) range. A useful interval extension is the **mean value form**

$$f(X) \subseteq F_{MV}(X) = f(m) + \sum_{i=1}^{n} D_i F(X)(X_i - m_i)$$

where X denotes an interval vector and m is the midpoint vector while $D_i F$ be an interval extension of the first derivatives $\partial f / \partial x_i$.

Vertex method, interval substitution, etc.

```
Propagation
```

5.2 handling with measurement uncertainty

Naturally, intervals serve as an intuitive representations for measurement error. Engineers often report measurement incertitude in the form of $[m \pm w]$. Many statistical models arise based on the interval statistics. As an example, similar to using a probability distribution to characterise precise data, we can employ a p-box to characterise interval-valued data. Generalisation of empirical cumulative distribution function can also be intuitively made. In addition, Kolmogorov–Smirnov bounds can also be generalised to derive confidence limits for imprecise data.

$$[\overline{F}(x), \underline{F}(x)] = \left[\min(1, \hat{F}_L(x) + D_N^{\alpha}),\ \max(0, \hat{F}_R(x) - D_N^{\alpha}) \right]$$

`PyUncertainNumber` provides a straghtforward syntax in charactersing interval-valued data.

```
characterisation
```

CHAPTER 6

API REFERENCE

This page contains auto-generated API reference documentation[1].

6.1 sampling

6.1.1 Functions

*index_to_bool_*(index[, dim])	Converts a vector of indices to an array of boolean pairs for masking.
sampling_method(...)	sampling of intervals

6.1.2 Module Contents

sampling.**index_to_bool_**(*index: numpy.ndarray*, *dim=2*)

Converts a vector of indices to an array of boolean pairs for masking.

Parameters

- **index** – A NumPy array of integer indices representing selected elements or categories. The values in *index* should be in the range [0, dim-1].
- **dim** (*scalar*) – The number of categories or dimensions in the output boolean array. Defaults to 2.

signature:

index_to_bool_(index:np.ndarray,dim=2) -> tuple

Note

- the augument *index* is an np.ndaray of the index of intervals.

[1] Created with sphinx-autoapi

- the argument *dim* will specify the function mapping of variables to be propagated.
- If dim > 2, e.g. (2,0,1,0) the array of booleans is [(0,0,1),(1,0,0),(0,1,0),(1,0,0)].

Returns

- A NumPy array of boolean pairs representing the mask.

sampling.**sampling_method**(*x: numpy.ndarray*, *f: Callable*, *results:* pyuncertainnumber.propagation.utils.Propagation_results = *None*, *n_sam: int = 500*, *method='monte_carlo'*, *save_raw_data='no'*, *endpoints=False*) → *pyuncertainnumber.propagation.utils.Propagation_results*

sampling of intervals

Parameters

- **x** (*np.ndarray*) – A 2D NumPy array where each row represents an input variable and the two columns define its lower and upper bounds (interval).
- **f** (*Callable*) – A callable function that takes a 1D NumPy array of input values and returns the corresponding output(s). Can be None, in which case only samples are generated.
- **n_sam** (*int*) – The number of samples to generate for the chosen sampling method.
- **method** (*str, optional*) –

  The sampling method to use. Choose from:

  - 'monte_carlo': Monte Carlo sampling (random sampling from uniform distributions)
  - 'latin_hypercube': Latin Hypercube sampling (stratified sampling for better space coverage)

  Defaults to 'monte_carlo'.
- **endpoints** (*bool, optional*) – If True, include the interval endpoints in the sampling. Defaults to False.
- **save_raw_data** (*str, optional*) – Whether to save raw data. Options: 'yes', 'no'. Defaults to 'no'.

signature:

sampling_method(x:np.ndarray, f:Callable, n_sam:int, method ='montecarlo', endpoints=False, results:dict = None, save_raw_data = 'no') -> dict of np.ndarrays

Note

- The function assumes that the na in *x* represent uniform distributions.
- If the *f* function returns multiple outputs, the *all_output* array will be 2-dimensional y and x for all x samples.

Returns

A dictionary containing the results:

- 'bounds': An np.ndarray of the bounds for each output parameter (if f is not None).
- **'min': A dictionary for lower bound results (if f in not None)**

- 'x': Input values that produced the miminum output value(s) (if f is not None).
- 'f': Minimum output value(s) (if f is not None).

- **'max': A dictionary for upper bound results (if f in not None)**
  - 'x': Input values that produced the maximum output value(s) (if f is not None).
  - 'f': Maximum output value(s) (if f is not None).
- **'raw_data': A dictionary containing raw data (if save_raw_data is 'yes'):**
  - 'x': All generated input samples.
  - 'f': Corresponding output values for each input sample.

Return type
dict

Example

```
>>> x = np.array([[1, 2], [3, 4], [5, 6]])  # Define input intervals
>>> f = lambda x: x[0] + x[1] + x[2]  # Define the function
>>> y = sampling_method(x, f, n_sam=500, method='monte_carlo', endpoints=False,␣
↪save_raw_data='no')
```

6.2 endpoints

6.2.1 Functions

endpoints_method(...)	Performs uncertainty propagation using the Endpoints Method. The function assumes that the intervals in *x* represent uncertainties

6.2.2 Module Contents

endpoints.**endpoints_method**(*x: numpy.ndarray*, *f: Callable*, *results:* pyuncertainnumber.propagation.utils.Propagation_results = *None*, *save_raw_data='no'*) → *pyuncertainnumber.propagation.utils.Propagation_results*

Performs uncertainty propagation using the Endpoints Method. The function assumes that the intervals in *x* represent uncertainties and aims to provide conservative bounds on the output uncertainty. If the *f* function returns multiple outputs, the *bounds* array will be 2-dimensional.

Parameters

- **x** (-) – A 2D NumPy array where each row represents an input variable and the two columns define its lower and upper bounds (interval).
- **f** (-) – A callable function that takes a 1D NumPy array of input values and returns the corresponding output(s).
- **save_raw_data** (-) – Controls the amount of data returned. - 'no': Returns only the minimum and maximum output values along with the

  corresponding input values.

– **'yes': Returns the above, plus the full arrays of unique input combinations** (*all_input*) and their corresponding output values (*all_output*).

signature:
endpoints_method(x:np.ndarray, f:Callable, save_raw_data = 'no') -> dict

Note

Example usage with different parameters for minimization and maximization f = lambda x: x[0] + x[1] + x[2] # Example function

Determine input parameters for function and method x_bounds = np.array([[1, 2], [3, 4], [5, 6]])

Returns

A dictionary containing the results:

- 'bounds': An np.ndarray of the bounds for each output parameter (if f is not None).
- 'min': A dictionary for lower bound results (if f is not None): - 'x': Input values that produced the minimum output value(s). - 'f': Minimum output value(s).
- 'max': A dictionary for upper bound results (if f is not None): - 'x': Input values that produced the maximum output value(s). - 'f': Maximum output value(s).
- 'raw_data': A dictionary containing raw data (if *save_raw_data* is 'yes'): - 'x': All generated input samples. - 'f': Corresponding output values for each input sample.

Return type

- dict

Example

```
>>> y = endpoints_method(x_bounds, f)
```

6.3 subinterval

6.3.1 Functions

subinterval_method(...)	subinterval reconstitution method

6.3.2 Module Contents

subinterval.**subinterval_method**(*x: numpy.ndarray*, *f: Callable*, *results:* pyuncertainnumber.propagation.utils.Propagation_results = *None*, *n_sub: numpy.array = 3*, *save_raw_data='no'*) → *pyuncertainnumber.propagation.utils.Propagation_results*

subinterval reconstitution method

Parameters

- **x** (-) – A 2D NumPy array where each row represents an input variable and the two columns define its lower and upper bounds (interval).

- **f** (-) – A callable function that takes a 1D NumPy array of input values and returns the corresponding output(s).
- **n_sub** (-) – A scalar (integer) or a 1D NumPy array specifying the number of subintervals for each input variable.
  - If a scalar, all input variables are divided into the same number of subintervals (defaults 3 divisions).
  - If an array, each element specifies the number of subintervals for the corresponding input variable.
- **save_raw_data** (-) – Controls the amount of data returned: - 'no': Returns only the minimum and maximum output values along with the

  corresponding input values.

  - **'yes': Returns the above, plus the full arrays of unique input combinations** (*all_input*) and their corresponding output values (*all_output*).

signature:

subinterval_method(x:np.ndarray, f:Callable, n:np.array, results:dict = None, save_raw_data = 'no') -> dict

Note

- **The function assumes that the intervals in *x* represent uncertainties and aims to provide conservative** bounds on the output uncertainty.
- **The computational cost increases exponentially with the number of input variables** and the number of subintervals per variable.
- If the *f* function returns multiple outputs, the *all_output* array will be 2-dimensional.

Returns

A dictionary containing the results:

- 'bounds': An np.ndarray of the bounds for each output parameter (if f is not None).

Return type

- dict
- **'min': A dictionary for lower bound results (if f is not None):**
  - 'x': Input values that produced the minimum output value(s).
  - 'f': Minimum output value(s).
- **'max': A dictionary for upper bound results (if f is not None):**
  - 'x': Input values that produced the maximum output value(s).
  - 'f': Maximum output value(s).
- **'raw_data': A dictionary containing raw data (if *save_raw_data* is 'yes'):**
  - 'x': All generated input samples.
  - 'f': Corresponding output values for each input sample.

Example

```
>>> #Define input intervals
>>> x = np.array([[1, 2], [3, 4], [5, 6]])
>>> # Define the function
>>> f = lambda x: x[0] + x[1] + x[2]
>>> # Run sampling method with n = 2
>>> y = subinterval_method(x, f, n_sub, save_raw_data = 'yes')
>>> # Print the results
>>> y.print()
```

6.4 extremepoints

6.4.1 Functions

extremepoints_method(...)	Performs uncertainty propagation using the Extreme Point Method for monotonic functions.

6.4.2 Module Contents

extremepoints.**extremepoints_method**(*x: numpy.ndarray*, *f: Callable*, *results:* pyuncertainnumber.propagation.utils.Propagation_results = *None*, *save_raw_data='no'*) → *pyuncertainnumber.propagation.utils.Propagation_results*

Performs uncertainty propagation using the Extreme Point Method for monotonic functions. This method estimates the bounds of a function's output by evaluating it at specific combinations of extreme values (lower or upper bounds) of the input variables. It is efficient for monotonic functions but might not be accurate for non-monotonic functions. If the f function returns multiple outputs, the *bounds* array will be 2-dimensional.

Parameters

- **x** (-) – A 2D NumPy array where each row represents an input variable and the two columns define its lower and upper bounds (interval).
- **f** (-) – A callable function that takes a 1D NumPy array of input values and returns the corresponding output(s).
- **save_raw_data** (-) – Controls the amount of data returned. - 'no': Returns only the minimum and maximum output values along with the

  corresponding input values.

  – **'yes': Returns the above, plus the full arrays of unique input combinations** (*all_input*) and their corresponding output values (*all_output*).

signature:
extremepoints_method(x:np.ndarray, f:Callable, results:dict, save_raw_data = 'no') -> dict

Returns

- A Propagation_results object containing the results. - 'bounds': An np.ndarray of the bounds for each output parameter (if f is not None). - 'sign_x': A NumPy array of shape (num_outputs, d) containing the signs (i.e., positive, negative)

used to determine the extreme points for each output.

- 'min': A dictionary for lower bound results (if f is not None): - 'x': Input values that produced the minimum output value(s). - 'f': Minimum output value(s).
- 'max': A dictionary for upper bound results (if f is not None): - 'x': Input values that produced the maximum output value(s). - 'f': Maximum output value(s).
- 'raw_data': A dictionary containing raw data (if *save_raw_data* is 'yes'): - 'x': All generated input samples. - 'f': Corresponding output values for each input sample.

Example

Example usage with different parameters for minimization and maximization >>> f = lambda x: x[0] + x[1] + x[2] # Example function >>> # Determine input parameters for function and method >>> x_bounds = np.array([[1, 2], [3, 4], [5, 6]]) >>> # Call the method >>> y = extremepoint_method(x_bounds, f) >>> # print results >>> y.print()

6.5 Taylor_series

6.6 endpoints_cauchy

6.6.1 Functions

cauchydeviates_method(...)	This method propagates intervals through a balck box model with the endpoint Cauchy deviate method. It is an approximate method, so the user should expect non-identical results for different runs.

6.6.2 Module Contents

endpoints_cauchy.**cauchydeviates_method**(*x: numpy.ndarray*, *f: Callable*, *results:* pyuncertainnumber.propagation.utils.Propagation_results = *None*, *n_sam: int = 500*, *save_raw_data='no'*) → *pyuncertainnumber.propagation.utils.Propagation_results*

This method propagates intervals through a balck box model with the endpoint Cauchy deviate method. It is an approximate method, so the user should expect non-identical results for different runs.

Parameters

- **x** (*np.ndarray*) – A 2D NumPy array representing the intervals for each input variable. Each row should contain two elements: the lower and upper bounds of the interval.
- **f** (*Callable*) – A callable function that takes a 1D NumPy array of input values and returns a single output value or an array of output values. Can be None, in which case only the Cauchy deviates (x) and the maximum Cauchy deviate (K) are returned.
- **results_class** (Propagation_results) – The class to use for storing results (defaults to Propagation_results).
- **n_sam** (*int*) – The number of samples (Cauchy deviates) to generate for each input variable (defaults 500 samples).
- **save_raw_data** (*str, optional*) – Whether to save raw data. Defaults to 'no'. Currently not supported by this method.

signature:

cauchydeviate_method(x: np.ndarray, f: Callable, results_class = Propagation_results,
n_sam: int, save_raw_data='no') -> dict

Returns

A PropagationResult object containing the results.

- **'raw_data': A dictionary containing raw data (if f is None):**
  - 'x': Cauchy deviates (x).
  - 'K': Maximum Cauchy deviate (K).
  - 'bounds': An np.ndarray of the bounds for each output parameter (if f is not None).
  - **'min': A dictionary for lower bound results (if f is not None).**
    - 'f': Minimum output value(s).
    - 'x': None (as input values corresponding to min/max are not tracked in this method).
  - **'max': A dictionary for upper bound results (if f is not None).**
    - 'f': Maximum output value(s).
    - 'x': None (as input values corresponding to min/max are not tracked in this method).

Example

```
>>> f = lambda x: x[0] + x[1] + x[2]  # Example function
>>> x_bounds = np.array([[1, 2], [3, 4], [5, 6]])
>>> y = cauchydeviates_method(x_bounds,f=f, n_sam=50, save_raw_data = 'yes')
```

6.7 pyuncertainnumber

6.7.1 Submodules

pyuncertainnumber.UV

pyuncertainnumber.characterisation

Submodules

pyuncertainnumber.characterisation.check

Classes

DistributionSpecification	an attempt to double check the user specification for a pbox or dist

Module Contents

class pyuncertainnumber.characterisation.check.**DistributionSpecification**(*dist_family: str*, *dist_params: tuple*)

an attempt to double check the user specification for a pbox or dist

> **Note**
>
> - canonical form: [‘gaussian’, ([0,1], [1,2])]

TODO: unfinished logic

dist_family

dist_params

tell_i_flag()

boolean about if imprecise specification

property i_flag

get_specification()

pyuncertainnumber.characterisation.core

Functions

makeUN(func)	return from construct a Uncertain Number object

Module Contents

pyuncertainnumber.characterisation.core.**makeUN**(*func*)

return from construct a Uncertain Number object

pyuncertainnumber.characterisation.ensemble

Classes

Ensemble	Create a collection of name/value pairs.

Module Contents

class pyuncertainnumber.characterisation.ensemble.**Ensemble**(**args*, ***kwds*)

Bases: enum.Enum

Create a collection of name/value pairs.

Example enumeration:

```
>>> class Color(Enum):
...     RED = 1
...     BLUE = 2
...     GREEN = 3
```

Access them by:

- attribute access:

```
>>> Color.RED
<Color.RED: 1>
```

- value lookup:

```
>>> Color(1)
<Color.RED: 1>
```

- name lookup:

```
>>> Color['RED']
<Color.RED: 1>
```

Enumerations can be iterated over, and know how many members they have:

```
>>> len(Color)
3
```

```
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

Methods can be added to enumerations, and members can have their own attributes – see the documentation for details.

repeated_measurements = 'repeated measurements'

flights = 'flights'

pressurisations = 'pressurisations'

temporal_steps = 'temporal steps'

spatial_sites = 'spatial sites'

manufactured_components = 'manufactured components'

customers = 'customers'

people = 'people'

households = 'households'

particular_population = 'particular population'

pyuncertainnumber.characterisation.measurand

Classes

Measurand	Create a collection of name/value pairs.

Module Contents

class pyuncertainnumber.characterisation.measurand.**Measurand**(**args*, ***kwds*)

Bases: enum.Enum

Create a collection of name/value pairs.

Example enumeration:

```
>>> class Color(Enum):
...     RED = 1
...     BLUE = 2
...     GREEN = 3
```

Access them by:

- attribute access:

```
>>> Color.RED
<Color.RED: 1>
```

- value lookup:

```
>>> Color(1)
<Color.RED: 1>
```

- name lookup:

```
>>> Color['RED']
<Color.RED: 1>
```

Enumerations can be iterated over, and know how many members they have:

```
>>> len(Color)
3
```

```
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

Methods can be added to enumerations, and members can have their own attributes – see the documentation for details.

count = 'count'

tally = 'tally'

unobservable_parameter = 'unobservable parameter'

probability = 'probability'

distribution = 'distribution'

range_ = 'range'

rank = 'rank'

pyuncertainnumber.characterisation.multiple_UN

Classes

Deck	multiple UN objects in a deck

Functions

make_many_intervals()

Module Contents

pyuncertainnumber.characterisation.multiple_UN.**make_many_intervals**()

class pyuncertainnumber.characterisation.multiple_UN.**Deck**

multiple UN objects in a deck

> **Note**
>
> - the deck is a list of UN objects, where each UN object is termed as a card;

cards: List[PyUncertainNumber.UC.UncertainNumber] = []

__repr__()

JSON_dump(*filename='./results/mulUN_data.json'*)

pyuncertainnumber.characterisation.stats

Attributes

mom
mle
smle

Functions

makedist(shape)	change return from sps.dist to Distribution objects
singleParamPattern(x, shape)	
fit(method, family, data)	top-level fit from data
MMbernoulli(x)	a first attempt to Maximum likelihood estimation for exponential distribution
MMbeta(x)	
MMbetabinomial(n, x)	
MMbinomial(x, n)	
MMchisquared(x)	
MMexponential(x)	
MMF(x)	
MMgamma(x)	
MMgeometric(x)	
MMgeometric(x)	
MMpascal(x)	
MMgumbel(x)	
MMextremevalue(x)	
MMlognormal(x)	
MMlaplace(x)	
MMdoubleexponential(x)	
MMlogistic(x)	
MMloguniform(x)	
MMnormal(x)	
MMgaussian(x)	
MMpareto(x)	
MMpoisson(x)	
MMpowerfunction(x)	

continues on next page

Table 1 – continued from previous page

MMt(x)	
MMstudent(x)	
MMuniform(x)	
MMrectangular(x)	
MMtriangular(x[, iters, dives])	
MLbernoulli(x)	
MLbeta(x)	
MLbetabinomial(x)	
MLbinomial(x)	# TODO to check
MLchisquared(x)	
MLexponential(x)	a standalone caller for exponential distribution with interval data (not in use yet)
MLF(x)	
MLgamma(x)	
MLgammaexponential(x)	
MLgeometric(x)	
MLgumbel(x)	
MLlaplace(x)	
MLlogistic(x)	
MLlognormal(x)	
MLloguniform(x)	
MLnegativebinomial(x)	
MLnormal(x)	
MLpareto(x)	
MLpoisson(x)	
MLpowerfunction(x)	
MLrayleigh(x)	

continues on next page

Table 1 – continued from previous page

MLstudent(x)
MLtriangular(x)
MLuniform(x)
sMLbernoulli(x)
sMLnormal(x)
sMLgaussian(x)
sMLexponential(x)
sMLpoisson(x)
sMLgeometric(x)
sMLgumbel(x)
sMLpascal(x)
sMLuniform(x)
sMLrectangular(x)
sMLpareto(x)
sMLlaplace(x)
sMLdoubleexponential(x)
sMLlognormal2(x)
sMLlognormal(x)
sMLloguniform(x)
sMLweibull(x[, shapeinterval])
sMLgamma(data)
MEminmax(min, max)
MEminmaxmean(min, max, mean)
MEmeansd(mean, sd)
MEminmean(min, mean)
MEdiscretemean(x, mu[, steps, iterations])

continues on next page

Table 1 – continued from previous page

MEquantiles(v, p)
MEdiscreteminmax(min, max)
MEmeanvar(mean, var)
MEminmaxmeansd(min, max, mean, sd)
MEmmms(min, max, mean, sd)
MEminmaxmeanvar(min, max, mean, var)
antweiler(x)
betapert(min, max, mode)
mnr(n[, many])
fermilnorm(x1, x2[, n, pr])
ferminorm(x1, x2[, n, pr])
approxksD95(n)
ks(x[, conf, min, max])
ferminormconfband(x1, x2, n[, pr, conf, bOt, tOp])
fermilnormconfband(x1, x2, n[, pr, conf, bOt, tOp])

Module Contents

pyuncertainnumber.characterisation.stats.**makedist**(*shape: str*)

change return from sps.dist to Distribution objects

pyuncertainnumber.characterisation.stats.**singleParamPattern**(*x*, *shape: str*)

pyuncertainnumber.characterisation.stats.**fit**(*method: str*, *family: str*, *data: numpy.ndarray*)

top-level fit from data

Parameters

- **method** (-) – method of fitting, e.g., {'mle' or 'mom'} 'entropy', 'pert', 'fermi', 'bayesian'
- **family** (-) – distribution family to be fitted
- **data** (-) – data to be fitted

Note

- supported family list can be found in xx.

Returns

- the return from the constructors below are *scipy.stats.dist* objects or *UN* objects depending on the decorator

Example

```
>>> pun.fit('mle', 'norm', np.random.normal(0, 1, 100))
```

pyuncertainnumber.characterisation.stats.**MMbernoulli**(*x*)

a first attempt to Maximum likelihood estimation for exponential distribution

which accepts both precise and imprecise data;
#! the example of *singleparam* pattern #! to change, add the 'interval_measurement' decorator .. note:

```
the attempt is successful per se, but not accommodating to the top-level calling␣
↪signature yet.

- precise data returns precise distrubution
- imprecise data need to be in Interval type to return a pbox
- interval data can return either a precise distribution or a pbox
```

pyuncertainnumber.characterisation.stats.**MMbeta**(*x: numpy.ndarray*)

pyuncertainnumber.characterisation.stats.**MMbetabinomial**(*n: int*, *x*)

pyuncertainnumber.characterisation.stats.**MMbinomial**(*x*, *n: int*)

Parameters

n (-) – number of trials

pyuncertainnumber.characterisation.stats.**MMchisquared**(*x*)

pyuncertainnumber.characterisation.stats.**MMexponential**(*x*)

pyuncertainnumber.characterisation.stats.**MMF**(*x*)

pyuncertainnumber.characterisation.stats.**MMgamma**(*x*)

pyuncertainnumber.characterisation.stats.**MMgeometric**(*x*)

pyuncertainnumber.characterisation.stats.**MMgeometric**(*x*)

pyuncertainnumber.characterisation.stats.**MMpascal**(*x*)

pyuncertainnumber.characterisation.stats.**MMgumbel**(*x*)

pyuncertainnumber.characterisation.stats.**MMextremevalue**(*x*)

pyuncertainnumber.characterisation.stats.**MMlognormal**(*x*)

pyuncertainnumber.characterisation.stats.**MMlaplace**(*x*)

pyuncertainnumber.characterisation.stats.**MMdoubleexponential**(*x*)

pyuncertainnumber.characterisation.stats.**MMlogistic**(*x*)

pyuncertainnumber.characterisation.stats.**MMloguniform**(*x*)

pyuncertainnumber.characterisation.stats.**MMnormal**(*x*)

pyuncertainnumber.characterisation.stats.**MMgaussian**(*x*)

pyuncertainnumber.characterisation.stats.**MMpareto**(*x*)

pyuncertainnumber.characterisation.stats.**MMpoisson**(*x*)

pyuncertainnumber.characterisation.stats.**MMpowerfunction**(*x*)

pyuncertainnumber.characterisation.stats.**MMt**(*x*)

pyuncertainnumber.characterisation.stats.**MMstudent**(*x*)

pyuncertainnumber.characterisation.stats.**MMuniform**(*x*)

pyuncertainnumber.characterisation.stats.**MMrectangular**(*x*)

pyuncertainnumber.characterisation.stats.**MMtriangular**(*x*, *iters=100*, *dives=10*)

pyuncertainnumber.characterisation.stats.**mom**

pyuncertainnumber.characterisation.stats.**MLbernoulli**(*x*)

pyuncertainnumber.characterisation.stats.**MLbeta**(*x*)

pyuncertainnumber.characterisation.stats.**MLbetabinomial**(*x*)

pyuncertainnumber.characterisation.stats.**MLbinomial**(*x*)

TODO to check #! no fitting func for scipy discrete distributions

pyuncertainnumber.characterisation.stats.**MLchisquared**(*x*)

pyuncertainnumber.characterisation.stats.**MLexponential**(*x*)

a standalone caller for exponential distribution with interval data (not in use yet)

pyuncertainnumber.characterisation.stats.**MLF**(*x*)

pyuncertainnumber.characterisation.stats.**MLgamma**(*x*)

pyuncertainnumber.characterisation.stats.**MLgammaexponential**(*x*)

pyuncertainnumber.characterisation.stats.**MLgeometric**(*x*)

pyuncertainnumber.characterisation.stats.**MLgumbel**(*x*)

pyuncertainnumber.characterisation.stats.**MLlaplace**(*x*)

pyuncertainnumber.characterisation.stats.**MLlogistic**(*x*)

pyuncertainnumber.characterisation.stats.**MLlognormal**(*x*)

pyuncertainnumber.characterisation.stats.**MLloguniform**(*x*)

pyuncertainnumber.characterisation.stats.**MLnegativebinomial**(*x*)

pyuncertainnumber.characterisation.stats.**MLnormal**(*x*)

pyuncertainnumber.characterisation.stats.**MLpareto**(*x*)

pyuncertainnumber.characterisation.stats.**MLpoisson**(*x*)

pyuncertainnumber.characterisation.stats.**MLpowerfunction**(*x*)

pyuncertainnumber.characterisation.stats.**MLrayleigh**(*x*)

pyuncertainnumber.characterisation.stats.**MLstudent**(*x*)

pyuncertainnumber.characterisation.stats.**MLtriangular**(*x*)

pyuncertainnumber.characterisation.stats.**MLuniform**(*x*)

pyuncertainnumber.characterisation.stats.**mle**

pyuncertainnumber.characterisation.stats.**sMLbernoulli**(*x*)

pyuncertainnumber.characterisation.stats.**sMLnormal**(*x*)

pyuncertainnumber.characterisation.stats.**sMLgaussian**(*x*)

pyuncertainnumber.characterisation.stats.**sMLexponential**(*x*)

pyuncertainnumber.characterisation.stats.**sMLpoisson**(*x*)

pyuncertainnumber.characterisation.stats.**sMLgeometric**(*x*)

pyuncertainnumber.characterisation.stats.**sMLgumbel**(*x*)

pyuncertainnumber.characterisation.stats.**sMLpascal**(*x*)

pyuncertainnumber.characterisation.stats.**sMLuniform**(*x*)

pyuncertainnumber.characterisation.stats.**sMLrectangular**(*x*)

pyuncertainnumber.characterisation.stats.**sMLpareto**(*x*)

pyuncertainnumber.characterisation.stats.**sMLlaplace**(*x*)

pyuncertainnumber.characterisation.stats.**sMLdoubleexponential**(*x*)

pyuncertainnumber.characterisation.stats.**sMLlognormal2**(*x*)

pyuncertainnumber.characterisation.stats.**sMLlognormal**(*x*)

pyuncertainnumber.characterisation.stats.**sMLloguniform**(*x*)

pyuncertainnumber.characterisation.stats.**sMLweibull**(*x*, *shapeinterval=None*)

pyuncertainnumber.characterisation.stats.**sMLgamma**(*data*)

pyuncertainnumber.characterisation.stats.**smle**

pyuncertainnumber.characterisation.stats.**MEminmax**(*min*, *max*)

pyuncertainnumber.characterisation.stats.**MEminmaxmean**(*min*, *max*, *mean*)

pyuncertainnumber.characterisation.stats.**MEmeansd**(*mean*, *sd*)

pyuncertainnumber.characterisation.stats.**MEminmean**(*min*, *mean*)

pyuncertainnumber.characterisation.stats.**MEdiscretemean**(*x*, *mu*, *steps=10*, *iterations=50*)

pyuncertainnumber.characterisation.stats.**MEquantiles**(*v*, *p*)

pyuncertainnumber.characterisation.stats.**MEdiscreteminmax**(*min*, *max*)

pyuncertainnumber.characterisation.stats.**MEmeanvar**(*mean*, *var*)

pyuncertainnumber.characterisation.stats.**MEminmaxmeansd**(*min*, *max*, *mean*, *sd*)

pyuncertainnumber.characterisation.stats.**MEmmms**(*min*, *max*, *mean*, *sd*)

pyuncertainnumber.characterisation.stats.**MEminmaxmeanvar**(*min*, *max*, *mean*, *var*)

pyuncertainnumber.characterisation.stats.**antweiler**(*x*)

pyuncertainnumber.characterisation.stats.**betapert**(*min*, *max*, *mode*)

pyuncertainnumber.characterisation.stats.**mnr**(*n*, *many=10000*)

pyuncertainnumber.characterisation.stats.**fermilnorm**(*x1*, *x2*, *n=None*, *pr=0.9*)

pyuncertainnumber.characterisation.stats.**ferminorm**(*x1*, *x2*, *n=None*, *pr=0.9*)

pyuncertainnumber.characterisation.stats.**approxksD95**(*n*)

pyuncertainnumber.characterisation.stats.**ks**(*x*, *conf=0.95*, *min=None*, *max=None*)

pyuncertainnumber.characterisation.stats.**ferminormconfband**(*x1*, *x2*, *n*, *pr=0.9*, *conf=0.95*, *bOt=0.001*, *tOp=0.999*)

pyuncertainnumber.characterisation.stats.**fermilnormconfband**(*x1*, *x2*, *n*, *pr=0.9*, *conf=0.95*, *bOt=0.001*, *tOp=0.999*)

pyuncertainnumber.characterisation.uncertainNumber

Classes

UncertainNumber	Uncertain Number class

Functions

_parse_interverl_inputs(vars)	Parse the input intervals

Module Contents

class pyuncertainnumber.characterisation.uncertainNumber.**UncertainNumber**

Uncertain Number class

Parameters

- **bounds;** (-)
- **distribution_parameters** (-) – a list of the distribution family and its parameters; e.g. ['norm', [0, 1]];
- **pbox_initialisation** (-) – a list of the distribution family and its parameters; e.g. ['norm', ([0,1], [3,4])];
- **naked_value** (-) – the deterministic numeric representation of the UN object, which shall be linked with the 'pba' or *Intervals* package

Example

```
>>> UncertainNumber(name="velocity", symbol="v", units="m/s", bounds=[1, 2])
```

name: str = None

symbol: str = None

units: Type[any] = None

_Q: Type[any] = None

uncertainty_type:
Type[*pyuncertainnumber.characterisation.uncertainty_types.Uncertainty_types*] = None

essence: str = None

masses: list[float] = None

bounds: List[float] | str = None

distribution_parameters: list[str, float | int] = None

pbox_parameters: list[str, Sequence[*pyuncertainnumber.pba.interval.Interval*]] =
None

hedge: str = None

_construct: Type[any] = None

naked_value: float = None

p_flag: bool = True

measurand: str = None

nature: str = None

provenence: str = None

justification: str = None

structure: str = None

security: str = None

ensemble: Type[*pyuncertainnumber.characterisation.ensemble.Ensemble*] = None

variability: str = None

dependence: str = None

uncertainty: str = None

instances = []

_samples: pyuncertainnumber.characterisation.utils.np.ndarray | list = None

parameterised_pbox_specification()

__post_init__()

the de facto initialisation method for the core math objects of the UN class

caveat:

user needs to by themselves figure out the correct shape of the 'distribution_parameters', such as ['uniform', [1,2]]

static match_pbox(*keyword*, *parameters*)

match the distribution keyword from the initialisation to create the underlying distribution object

Parameters

- **keyword** (-) – (str) the distribution keyword
- **parameters** (-) – (list) the parameters of the distribution

init_check()

check if the UN initialisation specification is correct

> **Note**
>
> a lot of things to double check. keep an growing list: 1. unit 2. hedge: user cannot speficy both 'hedge' and 'bounds'. 'bounds' takes precedence.

__str__()

the verbose user-friendly string representation .. note:

```
this has nothing to do with the logic of JSON serialisation
ergo, do whatever you fancy;
```

__repr__() → str

concise __repr__

describe(*type='verbose'*)

print out a verbose description of the uncertain number

_get_concise_representation()

get a concise representation of the UN object

ci()

get 95% range confidence interval

display(***kwargs*)

quick plot of the uncertain number object

property construct

classmethod from_hedge(*hedged_language*)

create an Uncertain Number from hedged language

> **Note**
>
> # if interval or pbox, to be implemented later on # currently only Interval is supported

classmethod fromConstruct(*construct*)

create an Uncertain Number from a construct object

classmethod fromDistribution(*D*, ***kwargs*)

create an Uncertain Number from specification of distribution

Parameters

- **D** (-) – Distribution object
- **dist_family** (*str*) – the distribution family
- **dist_params** (*list, tuple or string*) – the distribution parameters

classmethod from_Interval(*u*)

classmethod from_pbox(*p*)

genenal from pbox

classmethod from_ds(*ds*)

classmethod from_sps(*sps_dist*)

create an UN object from a parametric scipy.stats dist object #! it seems that a function will suffice :param - sps_dist: scipy.stats dist object

> **Note**
>
> - sps_dist –> UN.Distribution object

sqrt()

__add__(*other*)

add two uncertain numbers

__radd__(*other*)

__sub__(*other*)

__mul__(*other*)

multiply two uncertain numbers

__rmul__(*other*)

__truediv__(*other*)

divide two uncertain numbers

__rtruediv__(*other*)

__pow__(*other*)

power of two uncertain numbers

classmethod _toIntervalBackend(*vars=None*) → pyuncertainnumber.characterisation.utils.np.array

transform any UN object to an *interval* #! currently in use # TODO think if use Marco's Interval Vector object

question:

- what is the *interval* representation: list, nd.array or Interval object?

Returns

- 2D np.array representation for all the interval-typed UNs

classmethod _IntervaltoCompBackend(*vars*)

convert the interval-tupe UNs instantiated to the computational backend

> **Note**
>
> - it will automatically convert all the UN objects in array-like to the computational backend
> - essentially vars shall be all interval-typed UNs by now

Returns

- nd.array or Marco's Interval object

thoughts:

- if Marco's, then we'd use *intervalise* func to get all interval objects

and then to create another func to convert the interval objects to np.array to do endpoints method

JSON_dump(*filename='UN_data.json'*)

the JSON serialisation of the UN object into the filesystem

random(*size=None*)

Generate random samples from the distribution.

ppf(*q=None*)

"Calculate the percent point function (inverse of CDF) at quantile q.

pyuncertainnumber.characterisation.uncertainNumber.**_parse_interverl_inputs**(*vars*)

Parse the input intervals

> **Note**
>
> - Ioanna's funcs typically take 2D NumPy arra

pyuncertainnumber.characterisation.uncertainty_types

Classes

Uncertainty_types	Create a collection of name/value pairs.

Module Contents

class pyuncertainnumber.characterisation.uncertainty_types.**Uncertainty_types**(**args*, ***kwds*)

Bases: enum.Enum

Create a collection of name/value pairs.

Example enumeration:

```
>>> class Color(Enum):
...     RED = 1
...     BLUE = 2
...     GREEN = 3
```

Access them by:

- attribute access:

```
>>> Color.RED
<Color.RED: 1>
```

- value lookup:

```
>>> Color(1)
<Color.RED: 1>
```

- name lookup:

```
>>> Color['RED']
<Color.RED: 1>
```

Enumerations can be iterated over, and know how many members they have:

```
>>> len(Color)
3
```

```
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

Methods can be added to enumerations, and members can have their own attributes – see the documentation for details.

Certain = 'certain'

Aleatory = 'aleatory'

Epistemic = 'epistemic'

Inferential = 'inferential'

Design_uncertainty = 'design uncertainty'

Vagueness = 'vagueness'

Mixture = 'mixture'

pyuncertainnumber.characterisation.utils

Classes

EnhancedJSONEncoder	a template for jsonify general (dataclass) object
PBAEncoder	a bespoke JSON encoder for the PBA object
UNEncoder	a bespoke JSON encoder for the UncertainNumber object

Functions

tranform_ecdf(s[, display])	plot the CDF return the quantile
pl_pcdf(dist[, ax, title])	plot CDF from parametric distribution objects
pl_ecdf(s[, ax, return_value])	plot the empirical CDF given samples
to_database(dict_list, db_name, col_name)	
cd_root_dir([depth])	
initial_list_checking(text)	detects if a string representation of a list
bad_list_checking(text)	detects if a syntactically wrong specification of a list
PlusMinus_parser(txt)	
parser4(text)	
percentage_finder(txt)	
percentage_converter(txt)	convert a percentage into a float number
get_concise_repr(a_dict)	
array2list(a_dict)	convert an array from a dictionary into a list
entries_to_remove(remove_entries, the_dict)	

Module Contents

pyuncertainnumber.characterisation.utils.**tranform_ecdf**(*s*, *display=False*, ***kwargs*)

plot the CDF return the quantile

Parameters

s – sample

pyuncertainnumber.characterisation.utils.**pl_pcdf**(*dist: type[scipy.stats.rv_continuous | scipy.stats.rv_discrete]*, *ax=None*, *title=None*, ***kwargs*)

plot CDF from parametric distribution objects

pyuncertainnumber.characterisation.utils.**pl_ecdf**(*s*, *ax=None*, *return_value=False*, ***kwargs*)

plot the empirical CDF given samples

Parameters

s (*array-like*) – sample which can be either raw data or deviates as a representation of dist construct

pyuncertainnumber.characterisation.utils.**to_database**(*dict_list*, *db_name*, *col_name*)

pyuncertainnumber.characterisation.utils.**cd_root_dir**(*depth=0*)

pyuncertainnumber.characterisation.utils.**initial_list_checking**(*text*)

detects if a string representation of a list

pyuncertainnumber.characterisation.utils.**bad_list_checking**(*text*)

detects if a syntactically wrong specification of a list

pyuncertainnumber.characterisation.utils.**PlusMinus_parser**(*txt*)

pyuncertainnumber.characterisation.utils.**parser4**(*text*)

pyuncertainnumber.characterisation.utils.**percentage_finder**(*txt*)

pyuncertainnumber.characterisation.utils.**percentage_converter**(*txt*)

convert a percentage into a float number

> **Note**
>
> force only 1 percentage

class pyuncertainnumber.characterisation.utils.**EnhancedJSONEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Bases: `json.JSONEncoder`

a template for jsonify general (dataclass) object

#TODO Interval object in not json serializable

default(*o*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return super().default(o)
```

class pyuncertainnumber.characterisation.utils.**PBAEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Bases: `json.JSONEncoder`

a bespoke JSON encoder for the PBA object

default(*o*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return super().default(o)
```

class pyuncertainnumber.characterisation.utils.**UNEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `json.JSONEncoder`

a bespoke JSON encoder for the UncertainNumber object

Note

- Currently I'm treating the JSON data represent of a UN object

the same as the __repr__ method. But this can be changed later on to show more explicitly the strucutre of pbox or distribution # TODO prettify the JSON output to be explicit e.g. 'essence': 'interval', 'interval_initialisation': [2, 3] to shown as 'interval' with lower end and upper end distribution to shown as the type and parameters; e.g. 'distribution': 'normal', 'parameters': [2, 3]

default(*o*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return super().default(o)
```

pyuncertainnumber.characterisation.utils.**get_concise_repr**(*a_dict*)

pyuncertainnumber.characterisation.utils.**array2list**(*a_dict*)

convert an array from a dictionary into a list

pyuncertainnumber.characterisation.utils.**entries_to_remove**(*remove_entries*, *the_dict*)

pyuncertainnumber.characterisation.variability

Classes

Variability	Create a collection of name/value pairs.

Module Contents

class pyuncertainnumber.characterisation.variability.**Variability**(**args*, ***kwds*)

Bases: enum.Enum

Create a collection of name/value pairs.

Example enumeration:

```
>>> class Color(Enum):
...     RED = 1
...     BLUE = 2
...     GREEN = 3
```

Access them by:

- attribute access:

```
>>> Color.RED
<Color.RED: 1>
```

- value lookup:

```
>>> Color(1)
<Color.RED: 1>
```

- name lookup:

```
>>> Color['RED']
<Color.RED: 1>
```

Enumerations can be iterated over, and know how many members they have:

```
>>> len(Color)
3
```

```
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

Methods can be added to enumerations, and members can have their own attributes – see the documentation for details.

point_estimate = 'point estimate'

confidence = 'Confidence'

pyuncertainnumber.nlp

Submodules

pyuncertainnumber.nlp.language_parsing

Classes

ApproximatorRegCoefficients	A dataclass to store the regression coefficients of the approximator function

Functions

hedge_interpret(...)	interpret linguistic hedge words into UncertainNumber objects
parse_interval_expression(expression)	Parse the expression to interpret and return an Interval-type Uncertain Number object
decipher_zrf(num, d)	decipher the value of z, r, and f
decipher_d(x)	parse the decimal place d from a number
is_number(n)	check if a string is a number
count_sigfigs(→ int)	Count the number of significant figures in a number string
count_sig_digits_bias(number)	to count the bias for the getting the significant digits after the decimal point
findWholeWord(w)	Find a whole word in a string
whole_word_detect(word, string)	Detect if a whole word is in a string, return y or n

Module Contents

pyuncertainnumber.nlp.language_parsing.**hedge_interpret**(*hedge: str*, *return_type='interval'*) → *pyuncertainnumber.pba.interval.Interval* | *pyuncertainnumber.pba.pbox_base.Pbox*

interpret linguistic hedge words into UncertainNumber objects

Parameters

- **hedge** (*str*) – the hedge numerical expression to be interpreted
- **return_type** (*str*) – the type of object to be returned, either 'interval' or 'pbox'

Note

- the return can either be an interval or a pbox object

Example

```
>>> hedge_interpret("about 200", return_type="pbox")
```

pyuncertainnumber.nlp.language_parsing.**parse_interval_expression**(*expression*)

Parse the expression to interpret and return an Interval-type Uncertain Number object

Parameters

expression (*str*) – the flexible string desired by Scott to instantiate a Uncertain Number

caveat:
the expression needs to have space between the values and the operators, such as '[15 +- 10%]'

Returns
an Interval object

class pyuncertainnumber.nlp.language_parsing.**ApproximatorRegCoefficients**

A dataclass to store the regression coefficients of the approximator function

A: float

B: float

C: float

D: float

E: float

F: float

G: float

H: float

sigma: float

static lognormal(*m*, *s*)

_cp(*z*, *r*, *f*)

pyuncertainnumber.nlp.language_parsing.**decipher_zrf**(*num*, *d*)

decipher the value of z, r, and f

Parameters

- **num** (*float | int*) – a number parsed from the string
- **d** (*int*) – the decimal place of the last significant digit in the exemplar number

Returns
order of magnitude, defined to be the base-ten logoriathm of the exemplar number; r: roundness, defined as the $-d$ f: if the last digit is 5 or 0. If the last digit is 5, $f=1$, otherwise $f=0$

Return type
z

#TODO d can be inferred from the number itself

pyuncertainnumber.nlp.language_parsing.**decipher_d**(*x*)

parse the decimal place d from a number

pyuncertainnumber.nlp.language_parsing.**is_number**(*n*)

check if a string is a number .. note:

```
- If string is not a valid `float`,
- it'll raise `ValueError` exception
```

pyuncertainnumber.nlp.language_parsing.**count_sigfigs**(*numstr: str*) → int

Count the number of significant figures in a number string

pyuncertainnumber.nlp.language_parsing.**count_sig_digits_bias**(*number*)

to count the bias for the getting the significant digits after the decimal point

> **Note**
>
> to exclude the sig digits before the decimal point

pyuncertainnumber.nlp.language_parsing.**findWholeWord**(*w*)

Find a whole word in a string

> **Note**
>
> this returns the matched word, but not directly a boolean

pyuncertainnumber.nlp.language_parsing.**whole_word_detect**(*word*, *string*)

Detect if a whole word is in a string, return y or n

pyuncertainnumber.pba

Submodules

pyuncertainnumber.pba.aggregation

Attributes

makeUN	

Functions

stochastic_mixture(l_uns[, weights, display])	it could work for either Pbox, distribution, DS structure or Intervals
stacking(vec_interval, weights[, display, return_type])	stochastic mixture operation of Intervals with probability masses
mixture_pbox(l_pboxes[, weights, display])	
mixture_ds(l_ds[, display])	mixture operation for DS structure
mixture_cdf()	
imposition(*args)	Returns the imposition/intersection of the p-boxes in *args
envelope(...)	

Module Contents

pyuncertainnumber.pba.aggregation.**makeUN**

pyuncertainnumber.pba.aggregation.**stochastic_mixture**(*l_uns*, *weights=None*, *display=False*, ***kwargs*)

it could work for either Pbox, distribution, DS structure or Intervals

Parameters

- **l_un** (-) – list of uncertain numbers
- **weights** (-) – list of weights
- **display** (-) – boolean for plotting

TODO mix types later .. note:: - currently only accepts same type objects

pyuncertainnumber.pba.aggregation.**stacking**(*vec_interval: pyuncertainnumber.pba.interval.nInterval | intervals.Interval*, *weights*, *display=False*, *return_type='pbox'*)

stochastic mixture operation of Intervals with probability masses

Parameters

- **l_un** (-) – list of uncertain numbers
- **weights** (-) – list of weights
- **display** (-) – boolean for plotting
- **return_type** (-) – {'pbox' or 'ds' or 'bounds'}

Returns

- the left and right bound F in *cdf_bundlebounds* by default

but can choose to return a p-box

Note

- together the interval and masses, it can be deemed that all the inputs

required is jointly a DS structure

pyuncertainnumber.pba.aggregation.**mixture_pbox**(*l_pboxes*, *weights=None*, *display=False*)

pyuncertainnumber.pba.aggregation.**mixture_ds**(*l_ds*, *display=False*)

mixture operation for DS structure

pyuncertainnumber.pba.aggregation.**mixture_cdf**()

pyuncertainnumber.pba.aggregation.**imposition**(**args:* pyuncertainnumber.pba.pbox_base.Pbox | *pyuncertainnumber.pba.interval.nInterval* | *float* | *int*)

Returns the imposition/intersection of the p-boxes in *args

Parameters

mixed (*- UN objects to be*)

Returns

- Pbox

Note

- #TODO verfication needed for the base function *p1.imp(p2)*

pyuncertainnumber.pba.aggregation.**envelope**(**args: pyuncertainnumber.pba.interval.nInterval | pyuncertainnumber.pba.pbox_base.Pbox | float*) → pyuncertainnumber.pba.interval.nInterval | *pyuncertainnumber.pba.pbox_base.Pbox*

Allows the envelope to be calculated for intervals and p-boxes.

The envelope is the smallest interval/pbox that contains all values within the arguments.

Parameters:

`*args`: The arguments for which the envelope needs to be calculated. The arguments can be intervals, p-boxes, or floats.

Returns:

`Pbox|Interval`: The envelope of the given arguments, which can be an interval or a p-box.

Error

`ValueError`: If less than two arguments are given.

`TypeError`: If none of the arguments are intervals or p-boxes.

pyuncertainnumber.pba.cbox

Attributes

named_cbox
named_nextvalue

Functions

interval_measurements(func)	decorator for incorporating interval valued data
infer_cbox(→ pyuncertainnumber.pba.cbox_Leslie.Cbox)	top-level call signature to infer a c-box given data and family, plus rarely additional kwargs
infer_predictive_distribution(family, data, **args)	top-level call for the next value predictive distribution
CBbernoulli_p(x)	
CBbernoulli(x)	
CBbinomial_p(x, N)	cbox for Bionomial parameter
CBbinomial(x, N)	
nextvalue_binomialnp(x)	
parameter_binomialnp_n(x)	
parameter_binomialnp_p(x)	
CBpoisson_lambda(x)	
CBpoisson(x)	
CBexponential_lambda(x)	
CBexponential(x)	
cboxNormalMu_base(x)	base function for precise sample x
CBnormal_mu(x[, style])	
CBnormal_sigma(x)	
CBnormal(x)	
CBlognormal(x)	
CBlognormal_mu(x)	
CBlognormal_sigma(x)	
CBuniform_midpoint(x)	
CBuniform_width(x)	
CBuniform_minimum(x)	
CBuniform_maximum(x)	
CBuniform(x)	
CBnonparametric(x)	
CBnormal_meandifference(x1, x2)	
CBnormal_addconvolution(x, error)	

Module Contents

pyuncertainnumber.pba.cbox.**interval_measurements**(*func*)

decorator for incorporating interval valued data

pyuncertainnumber.pba.cbox.**infer_cbox**(*family: str*, *data*, ***args*) → *pyuncertainnumber.pba.cbox_Leslie.Cbox*

top-level call signature to infer a c-box given data and family, plus rarely additional kwargs

Notes

- data (list): a list of data samples, e.g. [2]
- additina kwargs such as N for binomial family

Example

```
>>> infer_cbox(''binomial', data=[2], N=10)
```

pyuncertainnumber.pba.cbox.**infer_predictive_distribution**(*family: str*, *data*, ***args*)

top-level call for the next value predictive distribution

pyuncertainnumber.pba.cbox.**CBbernoulli_p**(*x*)

pyuncertainnumber.pba.cbox.**CBbernoulli**(*x*)

pyuncertainnumber.pba.cbox.**CBbinomial_p**(*x*, *N*)

cbox for Bionomial parameter

Parameters

- **x** (*list or int*) – sample data as in a list of success or number of success or a single int as the number of success k
- **N** (*int*) – number of trials

> **Note**
>
> x[i] ~ binomial(N, p), for unknown p, x[i] is a nonnegative integer but x is a int number, it suggests the number of success as *k*.

Returns

cbox object

Return type

cbox

pyuncertainnumber.pba.cbox.**CBbinomial**(*x*, *N*)

pyuncertainnumber.pba.cbox.**nextvalue_binomialnp**(*x*)

pyuncertainnumber.pba.cbox.**parameter_binomialnp_n**(*x*)

pyuncertainnumber.pba.cbox.**parameter_binomialnp_p**(*x*)

pyuncertainnumber.pba.cbox.**CBpoisson_lambda**(*x*)

pyuncertainnumber.pba.cbox.**CBpoisson**(*x*)

pyuncertainnumber.pba.cbox.**CBexponential_lambda**(*x*)

pyuncertainnumber.pba.cbox.**CBexponential**(*x*)

pyuncertainnumber.pba.cbox.**cboxNormalMu_base**(*x*)

base function for precise sample x

pyuncertainnumber.pba.cbox.**CBnormal_mu**(*x*, *style='analytical'*)

Parameters

- **x** – (array-like) the sample data
- **style** – (str) the style of the output CDF, either 'analytical' or 'samples'
- **size** – (int) the discritisation size. meaning the no of ppf in analytical style and the no of MC samples in samples style

Returns

(array-like) the CDF of the normal distribution

Return type

CDF

pyuncertainnumber.pba.cbox.**CBnormal_sigma**(*x*)

pyuncertainnumber.pba.cbox.**CBnormal**(*x*)

pyuncertainnumber.pba.cbox.**CBlognormal**(*x*)

pyuncertainnumber.pba.cbox.**CBlognormal_mu**(*x*)

pyuncertainnumber.pba.cbox.**CBlognormal_sigma**(*x*)

pyuncertainnumber.pba.cbox.**CBuniform_midpoint**(*x*)

pyuncertainnumber.pba.cbox.**CBuniform_width**(*x*)

pyuncertainnumber.pba.cbox.**CBuniform_minimum**(*x*)

pyuncertainnumber.pba.cbox.**CBuniform_maximum**(*x*)

pyuncertainnumber.pba.cbox.**CBuniform**(*x*)

pyuncertainnumber.pba.cbox.**CBnonparametric**(*x*)

pyuncertainnumber.pba.cbox.**CBnormal_meandifference**(*x1*, *x2*)

pyuncertainnumber.pba.cbox.**CBnonparametric_deconvolution**(*x*, *error*)

pyuncertainnumber.pba.cbox.**named_cbox**

pyuncertainnumber.pba.cbox.**named_nextvalue**

pyuncertainnumber.pba.cbox_Leslie

a Cbox constructor by Leslie

Classes

Cbox	Confidence boxes (c-boxes) are imprecise generalisations of traditional confidence distributions

Functions

cbox_from_extredists(rvs[, shape, extre_bound_params])	define cbox via parameterised extreme bouding distrbution functions
cbox_from_pseudosamples(samples)	

Module Contents

class pyuncertainnumber.pba.cbox_Leslie.**Cbox**(**args*, *extre_bound_params=None*, ***kwargs*)

Bases: *pyuncertainnumber.pba.pbox_base.Pbox*

Confidence boxes (c-boxes) are imprecise generalisations of traditional confidence distributions

They have a different interpretation to p-boxes but rely on the same underlying mathematics. As such in pba-for-python c-boxes inhert most of their methods from Pbox.

Parameters

Pbox (*_type_*) – _description_

extre_bound_params = None

__repr__()

display(*parameter_name=None*, ***kwargs*)

default plotting function

ci(*c=0.95*, *alpha=None*, *beta=None*, *style='two-sided'*)

query the confidence interval at a given confidence level *c*

pyuncertainnumber.pba.cbox_Leslie.**cbox_from_extredists**(*rvs*, *shape=None*, *extre_bound_params=None*)

define cbox via parameterised extreme bouding distrbution functions

Parameters

- **rvs** (*list*) – list of *scipy.stats.rv_continuous* objects
- **extre_bound_params** (*list*) – list of parameters for the extreme bounding c.d.f

pyuncertainnumber.pba.cbox_Leslie.**cbox_from_pseudosamples**(*samples*)

pyuncertainnumber.pba.constructors

Functions

pbox_fromeF(a, b)	pbox from emipirical CDF bundle
pbox_from_extredists(rvs[, shape, extre_bound_params])	transform into pbox object from extreme bounds parameterised by *sps.dist*
pbox_from_pseudosamples(samples)	a tmp constructor for pbox/cbox from approximate solution of the confidence/next value distribution
interpolate_p(x, y)	interpolate the cdf bundle for discrete distribution or ds structure

Module Contents

pyuncertainnumber.pba.constructors.**pbox_fromeF**(*a:* pyuncertainnumber.pba.utils.CDF_bundle, *b:* pyuncertainnumber.pba.utils.CDF_bundle)

pbox from emipirical CDF bundle :param - a: CDF bundle of lower extreme F; :param - b: CDF bundle of upper extreme F;

pyuncertainnumber.pba.constructors.**pbox_from_extredists**(*rvs*, *shape='beta'*, *extre_bound_params=None*)

transform into pbox object from extreme bounds parameterised by *sps.dist*

Parameters

rvs (*list*) – list of scipy.stats.rv_continuous objects

pyuncertainnumber.pba.constructors.**pbox_from_pseudosamples**(*samples*)

a tmp constructor for pbox/cbox from approximate solution of the confidence/next value distribution

Parameters

samples (*nd.array*) – the approximate Monte Carlo samples of the confidence/next value distribution

> **Note**
>
> ecdf is estimted from the samples and bridge to pbox/cbox

pyuncertainnumber.pba.constructors.**interpolate_p**(*x*, *y*)

interpolate the cdf bundle for discrete distribution or ds structure .. note:

```
- x: probabilities
- y: quantiles
```

pyuncertainnumber.pba.copula

Classes

Copula

Functions

ClaGen(x[, t])	
ClaInv(x[, t])	
FGen(x[, s])	
FInv(x[, s])	
indep(x, y)	
perf(x, y)	
opp(x, y)	
Cla(x, y[, t])	
F(x, y[, s])	
Gau(x, y[, r])	
pi([steps])	
M([steps])	
W([steps])	
Frank([s, steps])	
Clayton([t, steps])	
Gaussian([r, steps])	

Module Contents

class pyuncertainnumber.pba.copula.**Copula**(*cdf=None*, *func=None*, *param=None*)

Bases: `object`

cdf = None

func = None

param = None

__repr__()

get_cdf(*x*, *y*)

get_mass(*x*, *y*)

show(*pn=50*, *fontsize=20*, *cols=cm.RdGy*)

showContour(*fontsize=20*, *cols=cm.coolwarm*)

pyuncertainnumber.pba.copula.**ClaGen**(*x*, *t=1*)

pyuncertainnumber.pba.copula.**ClaInv**(*x*, *t=1*)

pyuncertainnumber.pba.copula.**FGen**(*x*, *s=1*)

pyuncertainnumber.pba.copula.**FInv**(*x*, *s=1*)

pyuncertainnumber.pba.copula.**indep**(*x*, *y*)

pyuncertainnumber.pba.copula.**perf**(*x*, *y*)

pyuncertainnumber.pba.copula.**opp**(*x*, *y*)

pyuncertainnumber.pba.copula.**Cla**(*x*, *y*, *t=1*)

pyuncertainnumber.pba.copula.**F**(*x*, *y*, *s=1*)

pyuncertainnumber.pba.copula.**Gau**(*x*, *y*, *r=0*)

pyuncertainnumber.pba.copula.**pi**(*steps=200*)

pyuncertainnumber.pba.copula.**M**(*steps=200*)

pyuncertainnumber.pba.copula.**W**(*steps=200*)

pyuncertainnumber.pba.copula.**Frank**(*s=0*, *steps=200*)

pyuncertainnumber.pba.copula.**Clayton**(*t=1*, *steps=200*)

pyuncertainnumber.pba.copula.**Gaussian**(*r=0*, *steps=200*)

pyuncertainnumber.pba.core

Functions

sum(*args[, method])	Allows the sum to be calculated for intervals and p-boxes
mean(*args[, method])	Allows the mean to be calculated for intervals and p-boxes
mul(*args[, method])	
sqrt(a)	

Module Contents

pyuncertainnumber.pba.core.**sum**(**args: pyuncertainnumber.pba.interval.Union[list, tuple]*, *method='f'*)

Allows the sum to be calculated for intervals and p-boxes

Parameters:

`*args`: pboxes or intervals `method` (`f,i,o,p`): addition method to be used

Returns:

`Interval | Pbox`: sum of interval or pbox objects within `*args`

> **Note**
>
> If a list or tuple is given as the first argument, the elements of the list or tuple are used as arguments. If only one (non-list) argument is given, the argument is returned.

pyuncertainnumber.pba.core.**mean**(**args: pyuncertainnumber.pba.interval.Union[list, tuple]*, *method='f'*)

Allows the mean to be calculated for intervals and p-boxes

Parameters:

l : list of pboxes or intervals

method : pbox addition method to be used

Output:

`Interval | Pbox`: mean of interval or pbox objects within `*args`

> **Important**
>
> Implemented as
>
> ```
> >>> pba.sum(*args,method = method)/len(args)
> ```

pyuncertainnumber.pba.core.**mul**(**args*, *method=None*)

pyuncertainnumber.pba.core.**sqrt**(*a*)

pyuncertainnumber.pba.distributions

distribution constructs

Attributes

named_dists

Classes

Distribution	two signature for the distribution object, either a parametric specification or a nonparametric sample per se

Functions

bernoulli(p)
beta(a, b)
betabinomial2(size, v, w)

continues on next page

Table 2 – continued from previous page

betabinomial(size, v, w)
binomial(size, p)
chisquared(v)
delta(a)
exponential([rate, mean])
exponential1([mean])
F(df1, df2)
gamma(shape[, rate, scale])
gammaexponential(shape[, rate, scale])
geometric(m)
gumbel(loc, scale)
inversechisquared(v)
inversegamma(shape[, scale, rate])
laplace(a, b)
logistic(loc, scale)
lognormal(m, s)
lognormal2(mlog, slog)
loguniform_solve(m, v)
loguniform([min, max, minlog, maxlog, mean, std])
loguniform1(m, s)
negativebinomial(size, prob)
normal(m, s)
pareto(mode, c)
poisson(m)
powerfunction(b, c)
rayleigh(loc, scale)

continues on next page

Table 2 – continued from previous page

sawinconrad(min, mu, max)
student(v)
uniform(a, b)
triangular(min, mode, max)
histogram(x)
mixture(x[, w])
left(x)
right(x)
uniroot(f, a)

Module Contents

class pyuncertainnumber.pba.distributions.**Distribution**

two signature for the distribution object, either a parametric specification or a nonparametric sample per se

dist_family: str = None

dist_params: list[float] | Tuple[float, Ellipsis] = None

sample_data: list[float] | numpy.ndarray = None

__post_init__()

__repr__()

rep()

the dist object either sps dist or sample approximated or pbox dist

flag()

boolean flag for if the distribution is a parameterised distribution or not .. note:

```
- only parameterised dist can do sampling
- for non-parameterised sample-data based dist, next steps could be fitting
```

sample(*size*)

generate deviates from the distribution

make_naked_value()

one value representation of the distribution .. note:: - use mean for now;

display(***kwargs*)

display the distribution

_get_hint()

fit(*data*)

fit the distribution to the data

property naked_value

property hint

classmethod dist_from_sps(*dist: scipy.stats.rv_continuous | scipy.stats.rv_discrete*, *shape: str = None*)

to_pbox()

convert the distribution to a pbox .. note:

```
- this only works for parameteried distributions for now
- later on work with sample-approximated dist until `fit()`is implemented
```

pyuncertainnumber.pba.distributions.**bernoulli**(*p*)

pyuncertainnumber.pba.distributions.**beta**(*a*, *b*)

pyuncertainnumber.pba.distributions.**betabinomial2**(*size*, *v*, *w*)

pyuncertainnumber.pba.distributions.**betabinomial**(*size*, *v*, *w*)

pyuncertainnumber.pba.distributions.**binomial**(*size*, *p*)

pyuncertainnumber.pba.distributions.**chisquared**(*v*)

pyuncertainnumber.pba.distributions.**delta**(*a*)

pyuncertainnumber.pba.distributions.**exponential**(*rate=1*, *mean=None*)

pyuncertainnumber.pba.distributions.**exponential1**(*mean=1*)

pyuncertainnumber.pba.distributions.**F**(*df1*, *df2*)

pyuncertainnumber.pba.distributions.**gamma**(*shape*, *rate=1*, *scale=None*)

pyuncertainnumber.pba.distributions.**gammaexponential**(*shape*, *rate=1*, *scale=None*)

pyuncertainnumber.pba.distributions.**geometric**(*m*)

pyuncertainnumber.pba.distributions.**gumbel**(*loc*, *scale*)

pyuncertainnumber.pba.distributions.**inversechisquared**(*v*)

pyuncertainnumber.pba.distributions.**inversegamma**(*shape*, *scale=None*, *rate=None*)

pyuncertainnumber.pba.distributions.**laplace**(*a*, *b*)

pyuncertainnumber.pba.distributions.**logistic**(*loc*, *scale*)

pyuncertainnumber.pba.distributions.**lognormal**(*m*, *s*)

pyuncertainnumber.pba.distributions.**lognormal2**(*mlog*, *slog*)

pyuncertainnumber.pba.distributions.**loguniform_solve**(*m*, *v*)

pyuncertainnumber.pba.distributions.**loguniform**(*min=None*, *max=None*, *minlog=None*, *maxlog=None*, *mean=None*, *std=None*)

pyuncertainnumber.pba.distributions.**loguniform1**(*m*, *s*)

pyuncertainnumber.pba.distributions.**negativebinomial**(*size*, *prob*)

pyuncertainnumber.pba.distributions.**normal**(*m*, *s*)

pyuncertainnumber.pba.distributions.**pareto**(*mode*, *c*)

pyuncertainnumber.pba.distributions.**poisson**(*m*)

pyuncertainnumber.pba.distributions.**powerfunction**(*b*, *c*)

pyuncertainnumber.pba.distributions.**rayleigh**(*loc*, *scale*)

pyuncertainnumber.pba.distributions.**sawinconrad**(*min*, *mu*, *max*)

pyuncertainnumber.pba.distributions.**student**(*v*)

pyuncertainnumber.pba.distributions.**uniform**(*a*, *b*)

pyuncertainnumber.pba.distributions.**triangular**(*min*, *mode*, *max*)

pyuncertainnumber.pba.distributions.**histogram**(*x*)

pyuncertainnumber.pba.distributions.**mixture**(*x*, *w=None*)

pyuncertainnumber.pba.distributions.**left**(*x*)

pyuncertainnumber.pba.distributions.**right**(*x*)

pyuncertainnumber.pba.distributions.**uniroot**(*f*, *a*)

pyuncertainnumber.pba.distributions.**named_dists**

pyuncertainnumber.pba.ds

Constructors for Dempester-Shafer structures.

Classes

dempstershafer_element	
DempsterShafer	Class for Dempester-Shafer structures.

Functions

plot_DS_structure(vec_interval[, weights, offset, ax])	plot the intervals in a vectorised form

Module Contents

class pyuncertainnumber.pba.ds.**dempstershafer_element**

Bases: `tuple`

interval

mass

class pyuncertainnumber.pba.ds.**DempsterShafer**(*intervals*, *masses: list[float]*)

Class for Dempester-Shafer structures.

Parameters

- **pairs** (- *the intervals argument accepts wildcard vector intervals {list of list*)
- **Interval**
- **nInterval};** (*pairs of*)
- **masses** (-) – probability masses

_intrep

_intervals

_masses

_create_DSstructure()

property structure

property intervals

property masses

disassemble()

display(*style='box'*, ***kwargs*)

to_pbox()

classmethod from_dsElements(**ds_elements:* dempstershafer_element)

Create a Dempster-Shafer structure from a list of Dempster-Shafer elements.

pyuncertainnumber.pba.ds.**plot_DS_structure**(*vec_interval: list[*pyuncertainnumber.pba.interval.Interval | *intervals.Interval]*, *weights=None*, *offset=0.3*, *ax=None*, ***kwargs*)

plot the intervals in a vectorised form

Parameters

- **vec_interval** – vectorised interval objects
- **weights** – weights of the intervals
- **offset** – offset for display the weights next to the intervals

pyuncertainnumber.pba.imprecise

Functions

imprecise_ecdf(...)	empirical cdf for interval valued data

Module Contents

pyuncertainnumber.pba.imprecise.**imprecise_ecdf**(*s: intervals.Interval*) → tuple[*pyuncertainnumber.pba.utils.CDF_bundle*, *pyuncertainnumber.pba.utils.CDF_bundle*]

empirical cdf for interval valued data

Returns

- left and right cdfs
- pbox

pyuncertainnumber.pba.interval

Attributes

I	

Classes

Interval	An interval is an uncertain number for which only the endpoints are known, $x = [a, b]$.

Functions

PM(x, hw)	Create an interval centered around x with a half-width of hw.

Module Contents

class pyuncertainnumber.pba.interval.**Interval**(*left=None*, *right=None*)

An interval is an uncertain number for which only the endpoints are known, $x = [a, b]$. This is interpreted as x being between a and b but with no more information about the value of x.

Intervals embody epistemic uncertainty within PBA.

Creation

Intervals can be created using either of the following:

```
>>> pba.Interval(0,1)
Interval [0,1]
>>> pba.I(2,3)
Interval [2,3]
```

Tip

The shorthand `I` is an alias for `Interval`

Intervals can also be created from a single value ± half-width:

```
>>> pba.PM(0,1)
Interval [-1,1]
```

By default intervals are displayed as `Interval [a,b]` where `a` and `b` are the left and right endpoints respectively. This can be changed using the **`interval.pm_repr`_** and **`interval.lr_repr`_** functions.

Arithmetic

For two intervals [a,b] and [c,d] the following arithmetic operations are defined:

Addition

$[a,b] + [c,d] = [a+c, b+d]$

Subtraction

$[a,b] - [c,d] = [a-d, b-c]$

Multiplication

$[a,b] * [c,d] = [\min(ac,ad,bc,bd), \max(ac,ad,bc,bd)]$

Division

$[a,b]/[c,d] = [a,b] * \frac{1}{[c,d]} \equiv [\min(a/c, a/d, b/c, b/d), \max(a/c, a/d, b/c, b/d)]$

Alternative arithmetic methods are described in *interval.add*, *interval.sub*, *interval.mul*, *interval.div*.

_left = None

_right = None

property left

property right

__repr__() → str

__str__() → str

__format__(*format_spec: str*) → str

__iter__()

__len__()

__radd__(*left*)

__sub__(*other*)

__rsub__(*other*)

__neg__()

__mul__(*other*)

__rmul__(*other*)

__truediv__(*other*)

__rtruediv__(*other*)

__pow__(*other*)

__rpow__(*left*)

__lt__(*other*)

__rgt__(*other*)

__eq__(*other*)

==

__gt__(*other*)

>

__rlt__(*other*)

__ne__(*other*)

!=

__le__(*other*)

<=

__ge__(*other*)

__bool__()

__abs__()

__contains__(*other*)

add(*other*, *method=None*)

Adds the interval and another object together.

Args:

`other`: The interval or numeric value to be added. This value must be transformable into an Interval object.

Methods:

p - perfect arithmetic $[a, b] + [c, d] = [a + c, b + d]$

o - opposite arithmetic $[a, b] + [c, d] = [a + d, b + c]$

None, i, f - Standard interval arithmetic is used.

Returns:

`Interval`

__add__(*other*)

padd(*other*)

> **Warning**
>
> This method is deprecated. Use add(other, method='p') instead.

oadd(*other*)

> **Warning**
>
> This method is deprecated. Use add(other, method='o') instead.

sub(*other*, *method=None*)

Subtracts other from self.

Args:

`other`: The interval or numeric value to be subracted. This value must be transformable into an Interval object.

Methods:

`p`: perfect arithmetic $a + b = [a.left - b.left, a.right - b.right]$

`o`: opposite arithmetic $a + b = [a.left - b.right, a.right - b.left]$

None, i, f - Standard interval arithmetic is used.

Returns:

`Interval`

psub(*other*)

> **Warning**
>
> Depreciated use self.sub(other, method = 'p') instead

osub(*other*)

> **Warning**
>
> Depreciated use self.sub(other, method = 'o') instead

mul(*other*, *method=None*)

Multiplies self by other.

Args:

`other`: The interval or numeric value to be multiplied. This value must be transformable into an Interval object.

Methods:

`p`: perfect arithmetic $[a, b], [c, d] = [a * c, b * d]$

`o`: opposite arithmetic $[a, b], [c, d] = [a * d, b * c]$

None, i, f - Standard interval arithmetic is used.

Returns:

Interval: The result of the multiplication.

pmul(*other*)

> **Warning**
>
> Depreciated use self.mul(other, method = 'p') instead

omul(*other*)

> **Warning**
>
> Depreciated use self.mul(other, method = 'o') instead

div(*other*, *method=None*)

Divides self by other

If $0 \in other$ it returns a division by zero error

Args:

`other` (Interval or numeric): The interval or numeric value to be multiplied. This value must be transformable into an Interval object.

Methods:

`p`: perfect arithmetic $[a,b],[c,d] = [a*1/c, b*1/d]$

`o`: opposite arithmetic $[a,b],[c,d] = [a*1/d, b*1/c]$

`None`, `i`, `f` - Standard interval arithmetic is used.

> **Implementation**
>
> ```
> >>> self.add(1/other, method = method)
> ```

> **Error**
>
> If $0 \in [a,b]$ it returns a division by zero error

pdiv(*other*)

> **Warning**
>
> Depreciated use self.div(other, method = 'p') instead

odiv(*other*)

> **Warning**
>
> Depreciated use self.div(other, method = 'o') instead

recip()

Calculates the reciprocle of the interval.

Returns:
`Interval`: Equal to $[1/b, 1/a]$
Example:

```
>>> pba.Interval(2,4).recip()
Interval [0.25, 0.5]
```

Error

If $0 \in [a, b]$ it returns a division by zero error

equiv(*other:* Interval) → bool

Checks whether two intervals are equivalent.

Parameters:
`other`: The interval to check against.
Returns `True` **if**:
`self.left == other.right` and `self.right == other.right`

`False` otherwise.

Error

`TypeError`: If `other` is not an instance of `Interval`

See also

`is_same_as()`

Examples:

```
>>> a = Interval(0,1)
>>> b = Interval(0.5,1.5)
>>> c = I(0,1)
>>> a.equiv(b)
False
>>> a.equiv(c)
True
```

property lo

Returns:
`self.left`

Tip

This function is redundant but exists to match Pbox class for possible internal reasons.

property hi

Returns:
`self.right`

> **Tip**
>
> This function is redundant but exists to match Pbox class for possible internal reasons.

width() → float

Returns:
`float`: The width of the interval, $\text{right} - \text{left}$

Example:

```
>>> pba.Interval(0,3).width()
3
```

halfwidth() → float

Returns:
`float`: The half-width of the interval, $(\text{right} - \text{left})/2$

Example:

```
>>> pba.Interval(0,3).halfwidth()
1.5
```

> **Implementation**
>
> ```
> >>> self.width()/2
> ```

midpoint() → float

Returns the midpoint of the interval

> **Note**
>
> - this serves as the deterministic value representation of the interval, a.k.a. the naked value for an interval

Returns
The midpoint of the interval, $(\text{right} + \text{left})/2$

Return type
`float`

Example:

```
>>> pba.Interval(0,2).midpoint()
1.0
```

to_logical()

Turns the interval into a logical interval, this is done by chacking the truth value of the ends of the interval

Returns:
`Logical`: The logical interval

> **Implementation**

```
>>> left = self.left.__bool__()
>>> right = self.right.__bool__()
>>> Logical(left,right)
```

env(*other: list* | Interval) → *Interval*

Calculates the envelope between two intervals

Parameters:

`other` : Interval or list. The interval to envelope with self

Hint

If other is a list then the envelope is calculated between self and each element of the list. In this case the envelope is calculated recursively and pba.envelope() may be more efficient.

Important

If other is a Pbox then `Pbox.env()` is called

See also

`pba.core.envelope`_

`pba.pbox.Pbox.env`_

Returns:

`Interval`: The envelope of self and other

straddles(*N: int* | *float* | Interval, *endpoints: bool = True*) → bool

Parameters:

`N`: Number to check. If N is an interval checks whether the whole interval is within self.

`endpoints`: Whether to include the endpoints within the check

Returns `True` **if**:

left $\leq N \leq$ right (Assuming `endpoints=True`).

For interval values. left $\leq N.left \leq$ right and left $\leq N.right \leq$ right (Assuming `endpoints=True`).

`False` otherwise.

Tip

`N in self` is equivalent to `self.straddles(N)`

straddles_zero(*endpoints=True*)

Checks whether 0 is within the interval

Implementation

Equivalent to `self.straddles(0,endpoints)`

See also

interval.straddles

intersection(*other:* Interval | *list*) → *Interval*

Calculates the intersection between intervals

Parameters:

`other`: The interval to intersect with self. If an interval is not given will try to cast as an interval. If a list is given will calculate the intersection between self and each element of the list.

Returns:

`Interval`: The intersection of self and other. If no intersection is found returns `None`

Example:

```
>>> a = Interval(0,1)
>>> b = Interval(0.5,1.5)
>>> a.intersection(b)
Interval [0.5, 1]
```

exp()

log()

sqrt()

display(*title=''*, *ax=None*, *style='band'*, ***kwargs*)

sample(*seed=None*, *numpy_rng: numpy.random.Generator = None*) → float

Generate a random sample within the interval.

Parameters:

`seed` (int, optional): Seed value for random number generation. Defaults to None.

`numpy_rng` (numpy.random.Generator, optional): Numpy random number generator. Defaults to None.

Returns:

`float`: Random sample within the interval.

Implementation

If `numpy_rng` is given:

```
>>> numpy_rng.uniform(self.left, self.right)
```

Otherwise the following is used:

```
>>> import random
>>> random.seed(seed)
>>> self.left + random.random() * self.width()
```

Examples:

```
>>> pba.Interval(0,1).sample()
0.6160988752201705
>>> pba.I(0,1).sample(seed = 1)
0.13436424411240122
```

If a numpy random number generator is given then it is used instead of the default python random number generator. It has to be initialised first.

```
>>> import numpy as np
>>> rng = np.random.default_rng(seed = 0)
>>> pba.I(0,1).sample(numpy_rng = rng)
0.6369616873214543
```

round()

outward rounding operation for an interval object

classmethod from_midwith(*midpoint: float*, *halfwidth: float*) → *Interval*

Creates an Interval object from a midpoint and half-width.

Parameters

- **midpoint** (-) – The midpoint of the interval.
- **halfwidth** (-) – The half-width of the interval.

Returns

The interval with midpoint and half-width.

Return type

- Interval

Example

```
>>> pba.Interval.from_midwith(0,1)
Interval [-1, 1]
```

pyuncertainnumber.pba.interval.**I**

pyuncertainnumber.pba.interval.**PM**(*x*, *hw*)

Create an interval centered around x with a half-width of hw.

TODO: a constructor for creating an interval from a midpoint and half-width # ! this func is weird and I keep it only for not breaking the code in other places.

Parameters:

x (float): The center value of the interval.

hw (float): The half-width of the interval.

Returns:

Interval: An interval object with lower bound x-hw and upper bound x+hw.

Error

ValueError: If hw is less than 0.

Example

```
>>> pba.PM(0, 1)
Interval [-1, 1]
```

pyuncertainnumber.pba.intervalOperators

Functions

parse_bounds(bounds)	parse the self.bounds argument
_str(bounds)	
wc_interval(bound)	wildcard scalar interval
_arraylike(x)	
_marco_interval_like(bound)	
_nick_interval_like(bound)	
make_vec_interval(vec)	vector interval implementation tmp
mean(x)	
_arraylike(x)	
_intervallike(x)	
std()	
var()	
roundInt()	outward rounding to integer

Module Contents

pyuncertainnumber.pba.intervalOperators.**parse_bounds**(*bounds*)

parse the self.bounds argument

pyuncertainnumber.pba.intervalOperators.**_str**(*bounds: str*)

pyuncertainnumber.pba.intervalOperators.**wc_interval**(*bound*)

wildcard scalar interval

pyuncertainnumber.pba.intervalOperators.**_arraylike**(*bound: list*)

pyuncertainnumber.pba.intervalOperators.**_marco_interval_like**(*bound: intervals.Interval*)

pyuncertainnumber.pba.intervalOperators.**_nick_interval_like**(*bound:* pyuncertainnumber.pba.interval.Interval)

pyuncertainnumber.pba.intervalOperators.**make_vec_interval**(*vec*)

vector interval implementation tmp

pyuncertainnumber.pba.intervalOperators.**mean**(*x*)

pyuncertainnumber.pba.intervalOperators.**_arraylike**(*x*)

pyuncertainnumber.pba.intervalOperators.**_intervallike**(*x*)

pyuncertainnumber.pba.intervalOperators.**std**()

pyuncertainnumber.pba.intervalOperators.**var**()

pyuncertainnumber.pba.intervalOperators.**roundInt**()

outward rounding to integer

pyuncertainnumber.pba.logical

Classes

Logical	Represents a logical value that can be either True or False or dunno ([False,True]).

Functions

is_same_as(a, b[, deep, exact_pbox])	Check if two objects of type 'Pbox' or 'Interval' are equal.
always(→ bool)	Checks whether the logical value is always true. i.e. Every value from one interval or p-box is always greater than any other values from another.
never(→ bool)	Checks whether the logical value is always true. i.e. Every value from one interval or p-box is always less than any other values from another.
sometimes(→ bool)	Checks whether the logical value is sometimes true. i.e. There exists one value from one interval or p-box is less than a values from another.
xtimes(→ bool)	Checks whether the logical value is exclusively sometimes true. i.e. There exists one value from one interval or p-box is less than a values from another but it is not always the case.

Module Contents

class pyuncertainnumber.pba.logical.**Logical**(*left: bool*, *right: bool = None*)

Bases: *pyuncertainnumber.pba.interval.Interval*

Represents a logical value that can be either True or False or dunno ([False,True]).

Inherits from the Interval class.

Attributes:

`left` (`bool`): The left endpoint of the logical value.

`right` (`bool`): The right endpoint of the logical value.

__bool__()

__repr__()

__str__

__invert__()

pyuncertainnumber.pba.logical.**is_same_as**(*a:* pyuncertainnumber.pba.pbox_base.Pbox | pyuncertainnumber.pba.interval.Interval, *b:* pyuncertainnumber.pba.pbox_base.Pbox | pyuncertainnumber.pba.interval.Interval, *deep=False*, *exact_pbox=True*)

Check if two objects of type 'Pbox' or 'Interval' are equal.

Parameters:

a: The first object to be compared.

b: The second object to be compared.

deep: If True, performs a deep comparison, considering object identity. If False, performs a shallow comparison based on object attributes. Defaults to False.

exact_pbox: If True, performs a deep comparison of p-boxes, considering all attributes. If False, performs a shallow comparison of p-boxes, considering only the left and right attributes. Defaults to True.

Returns True **if**:

bool: True if the objects have identical parameters. For Intervals this means that left and right are the same for both a and b. For p-boxes checks whether all p-box attributes are the same. If deep is True, checks whether the objects have the same id.

Examples:

```
>>> a = Interval(0, 2)
>>> b = Interval(0, 2)
>>> c = Interval(1, 3)
>>> is_same_as(a, b)
True
>>> is_same_as(a, c)
False
```

For p-boxes:

```
>>> a = pba.N([0,1],1)
>>> b = pba.N([0,1],1)
>>> c = pba.N([0,1],2)
>>> is_same_as(a, b)
True
>>> is_same_as(a, c)
False
>>> e = pba.box(0,1,steps=2)
>>> f = Pbox(left = [0,0],right=[1,1],steps=2)
>>> is_same_as(e, f, exact_pbox = True)
False
>>> is_same_as(e, f, exact_pbox = False)
True
```

pyuncertainnumber.pba.logical.**always**(*logical:* Logical | pyuncertainnumber.pba.interval.Interval | *numbers.Number* | *bool*) → bool

Checks whether the logical value is always true. i.e. Every value from one interval or p-box is always greater than any other values from another.

This function takes either a Logical object, an interval or a float as input and checks if both the left and right attributes of the Logical object are True. If an interval is provided, it checks that both the left and right attributes

of the Logical object are 1. If a numeric value is provided, it checks if the is equal to 1.

Parameters:

`logical` (`Logical`, `Interval` , `Number`): An object representing a logical condition with 'left' and 'right' attributes, or a number between 0 and 1.

Returns:

`bool`: True if both sides of the logical condition are True or if the float value is equal to 1, False otherwise.

Error

`TypeError`: If the input is not an instance of Interval, Logical or a numeric value.

`ValueError`: If the input float is not between 0 and 1 or the interval contains values outside of [0,1]

Examples:

```
>>> a = Interval(0, 2)
>>> b = Interval(1, 3)
>>> c = Interval(4, 5)
>>> always(a < b)
False
>>> always(a < c)
True
```

pyuncertainnumber.pba.logical.**never**(*logical:* Logical) → bool

Checks whether the logical value is always true. i.e. Every value from one interval or p-box is always less than any other values from another.

This function takes either a Logical object, an interval or a float as input and checks if both the left and right attributes of the Logical object are False. If an interval is provided, it checks that both the left and right attributes of the Logical object are 0. If a numeric value is provided, it checks if the is equal to 0.

Parameters:

`logical` (`Logical`, `Interval` , `Number`): An object representing a logical condition with 'left' and 'right' attributes, or a number between 0 and 1.

Returns:

`bool`: True if both sides of the logical condition are True or if the float value is equal to 0, False otherwise.

Error

`TypeError`: If the input is not an instance of Interval, Logical or a numeric value.

`ValueError`: If the input float is not between 0 and 1 or the interval contains values outside of [0,1]

Examples:

```
>>> a = Interval(0, 2)
>>> b = Interval(1, 3)
>>> c = Interval(4, 5)
>>> never(a < b)
False
>>> never(a < c)
True
```

pyuncertainnumber.pba.logical.**sometimes**(*logical:* Logical) → bool

Checks whether the logical value is sometimes true. i.e. There exists one value from one interval or p-box is less than a values from another.

This function takes either a Logical object, an interval or a float as input and checks if either the left and right attributes of the Logical object are True. If an interval is provided, it that both endpoints are not 0. If a numeric value is provided, it checks if the is not equal to 0.

Parameters:
`logical` (`Logical`, `Interval` , `Number`): An object representing a logical condition with 'left' and 'right' attributes, or a number between 0 and 1.

Returns:
`bool`: True if both sides of the logical condition are True or if the float value is equal to 0, False otherwise.

Error

`TypeError`: If the input is not an instance of Interval, Logical or a numeric value.

`ValueError`: If the input float is not between 0 and 1 or the interval contains values outside of [0,1]

Examples:

```
>>> a = pba.Interval(0, 2)
>>> b = pba.Interval(1, 4)
>>> c = pba.Interval(3, 5)
>>> pba.sometimes(a < b)
True
>>> pba.sometimes(a < c)
True
>>> pba.sometimes(c < b)
True
```

pyuncertainnumber.pba.logical.**xtimes**(*logical:* Logical) → bool

Checks whether the logical value is exclusively sometimes true. i.e. There exists one value from one interval or p-box is less than a values from another but it is not always the case.

This function takes either a Logical object, an interval or a float as input and checks that the left value is False and the right value is True If an interval is provided, it that both endpoints are not 0 or 1. If a numeric value is provided, it checks if the is not equal to 0 or 1.

Parameters:
`logical` (`Logical`, `Interval` , `Number`): An object representing a logical condition with 'left' and 'right' attributes, or a number between 0 and 1.

Returns:
`bool`: True if both sides of the logical condition are True or if the float value is equal to 0, False otherwise.

Error

`TypeError`: If the input is not an instance of Interval, Logical or a numeric value.

`ValueError`: If the input float is not between 0 and 1 or the interval contains values outside of [0,1]

Examples:

```
>>> a = pba.Interval(0, 2)
>>> b = pba.Interval(2, 4)
>>> c = pba.Interval(2.5,3.5)
>>> pba.xtimes(a < b)
False
>>> pba.xtimes(a < c)
False
>>> pba.xtimes(c < b)
True
```

pyuncertainnumber.pba.operation

Functions

convert(un)	transform the input un into a Pbox object

Module Contents

pyuncertainnumber.pba.operation.**convert**(*un*)

transform the input un into a Pbox object

> **Note**
>
> - theorically 'un' can be {Interval, DempsterShafer, Distribution, float, int}

pyuncertainnumber.pba.params

Hyperparameters for the pba

Classes

Params
Data
Named

Module Contents

class pyuncertainnumber.pba.params.**Params**

hedge_cofficients

steps = 200

many = 2000

p_values

p_lboundary = 0.0001

p_hboundary = 0.9999

scott_hedged_interpretation

user_hedged_interpretation

result_path = './results/'

hw = 0.5

class pyuncertainnumber.pba.params.Data

skinny = [[1.0, 1.52], [2.68, 2.98], [7.52, 7.67], [7.73, 8.35], [9.44, 9.99], [3.66, 4.58]]

puffy = [[3.5, 6.4], [6.9, 8.8], [6.1, 8.4], [2.8, 6.7], [3.5, 9.7], [6.5, 9.9], [0.15, 3.8], [4.5,...

sudret = [4.02, 4.07, 4.25, 4.32, 4.36, 4.45, 4.47, 4.57, 4.58, 4.62, 4.68, 4.71, 4.72, 4.79, 4.85, 4.86,...

k = 22

m = 11

n = 33

fdata

bdata

idata

data

x2

error

class pyuncertainnumber.pba.params.Named

k = 22

m = 11

n = 33

pyuncertainnumber.pba.pbox

Attributes

normal	
N	
gaussian	
U	
lognorm	
named_pbox	

Functions

_get_bounds(dist_family, *args[, steps])	from distribution specification to define the lower and upper bounds of the p-box
_bound_pcdf(dist_family, *args[, steps])	bound the parametric CDF
makePbox(func)	
norm(*args)	
lognormal(mean, var[, steps])	Creates a p-box for the lognormal distribution
alpha(*args)	
anglit(*args)	
argus(*args)	
arcsine(*args)	
beta(*args[, steps])	Beta distribution
betaprime(*args)	
bradford(*args)	
burr(*args)	
burr12(*args)	
cauchy(*args)	
chi(*args)	
chi2(*args)	

continues on next page

Table 3 – continued from previous page

cosine(*args)
crystalball(*args)
dgamma(*args)
dweibull(*args)
erlang(*args)
expon(*args)
exponnorm(*args)
exponweib(*args)
exponpow(*args)
f(*args)
fatiguelife(*args)
fisk(*args)
foldcauchy(*args)
foldnorm(mu, s[, steps])
genlogistic(*args)
gennorm(*args)
genpareto(*args)
genexpon(*args)
genextreme(*args)
gausshyper(*args)
gamma(*args)
gengamma(*args)
genhalflogistic(*args)
geninvgauss(*args)
gompertz(*args)
gumbel_r(*args)

continues on next page

Table 3 – continued from previous page

gumbel_l(*args)
halfcauchy(*args)
halflogistic(*args)
halfnorm(*args)
halfgennorm(*args)
hypsecant(*args)
invgamma(*args)
invgauss(*args)
invweibull(*args)
irwinhall(*args)
jf_skew_t(*args)
johnsonsb(*args)
johnsonsu(*args)
kappa4(*args)
kappa3(*args)
ksone(*args)
kstwo(*args)
kstwobign(*args)
laplace(*args)
laplace_asymmetric(*args)
levy(*args)
levy_l(*args)
levy_stable(*args)
logistic(*args)
loggamma(*args)
loglaplace(*args)

continues on next page

Table 3 – continued from previous page

loguniform(*args)
lomax(*args)
maxwell(*args)
mielke(*args)
moyal(*args)
nakagami(*args)
ncx2(*args)
ncf(*args)
nct(*args)
norminvgauss(*args)
pareto(*args)
pearson3(*args)
powerlaw(*args)
powerlognorm(*args)
powernorm(*args)
rdist(*args)
rayleigh(*args)
rel_breitwigner(*args)
rice(*args)
recipinvgauss(*args)
semicircular(*args)
skewcauchy(*args)
skewnorm(*args)
studentized_range(*args)
t(*args)
trapezoid(*args)

continues on next page

Table 3 – continued from previous page

triang(*args)	
truncexpon(*args)	
truncnorm(left, right[, mean, stddev, steps])	
truncpareto(*args)	
truncweibull_min(*args)	
tukeylambda(*args)	
uniform_sps(*args)	
vonmises(*args)	
vonmises_line(*args)	
wald(*args)	
weibull_min(*args)	
weibull_max(*args)	
wrapcauchy(*args)	
trapz(a, b, c, d[, steps])	
truncnorm(left, right[, mean, stddev, steps])	
uniform(a, b[, steps])	special case of Uniform distribution as
weibull(*args[, steps])	
KM(k, m[, steps])	
KN(k, n[, steps])	
bernoulli(*args)	
betabinom(*args)	
betanbinom(*args)	
binom(*args)	
boltzmann(*args)	
dlaplace(*args)	
geom(*args)	

continues on next page

Table 3 – continued from previous page

hypergeom(*args)
logser(*args)
nbinom(*args)
nchypergeom_fisher(*args)
nchypergeom_wallenius(*args)
nhypergeom(*args)
planck(*args)
poisson(*args)
randint(*args)
skellam(*args)
yulesimon(*args)
zipf(*args)
zipfian(*args)

Module Contents

pyuncertainnumber.pba.pbox.**_get_bounds**(*dist_family*, **args*, *steps=Params.steps*)

from distribution specification to define the lower and upper bounds of the p-box

Parameters

dist_family (-) – (str) the name of the distribution

pyuncertainnumber.pba.pbox.**_bound_pcdf**(*dist_family*, **args*, *steps=Params.steps*)

bound the parametric CDF

> **Note**
>
> - only support fully bounded parameters

pyuncertainnumber.pba.pbox.**makePbox**(*func*)

pyuncertainnumber.pba.pbox.**norm**(**args*)

pyuncertainnumber.pba.pbox.**lognormal**(*mean*, *var*, *steps=Params.steps*)

Creates a p-box for the lognormal distribution

Note: the parameters used are the mean and variance of the lognormal distribution not the mean and variance of the underlying normal See: *[1]<https://en.wikipedia.org/wiki/Log-normal_distribution#Generation_and_parameters>*

[2]<https://stackoverflow.com/questions/51906063/distribution-mean-and-standard-deviation-using-scipy-stats>

Parameters

- **mean** – mean of the lognormal distribution
- **var** – variance of the lognormal distribution

Return type

Pbox

pyuncertainnumber.pba.pbox.**alpha**(**args*)

pyuncertainnumber.pba.pbox.**anglit**(**args*)

pyuncertainnumber.pba.pbox.**argus**(**args*)

pyuncertainnumber.pba.pbox.**arcsine**(**args*)

pyuncertainnumber.pba.pbox.**beta**(**args*, *steps=Params.steps*)

Beta distribution

pyuncertainnumber.pba.pbox.**betaprime**(**args*)

pyuncertainnumber.pba.pbox.**bradford**(**args*)

pyuncertainnumber.pba.pbox.**burr**(**args*)

pyuncertainnumber.pba.pbox.**burr12**(**args*)

pyuncertainnumber.pba.pbox.**cauchy**(**args*)

pyuncertainnumber.pba.pbox.**chi**(**args*)

pyuncertainnumber.pba.pbox.**chi2**(**args*)

pyuncertainnumber.pba.pbox.**cosine**(**args*)

pyuncertainnumber.pba.pbox.**crystalball**(**args*)

pyuncertainnumber.pba.pbox.**dgamma**(**args*)

pyuncertainnumber.pba.pbox.**dweibull**(**args*)

pyuncertainnumber.pba.pbox.**erlang**(**args*)

pyuncertainnumber.pba.pbox.**expon**(**args*)

pyuncertainnumber.pba.pbox.**exponnorm**(**args*)

pyuncertainnumber.pba.pbox.**exponweib**(**args*)

pyuncertainnumber.pba.pbox.**exponpow**(**args*)

pyuncertainnumber.pba.pbox.**f**(**args*)

pyuncertainnumber.pba.pbox.**fatiguelife**(**args*)

pyuncertainnumber.pba.pbox.**fisk**(**args*)

pyuncertainnumber.pba.pbox.**foldcauchy**(**args*)

pyuncertainnumber.pba.pbox.**foldnorm**(*mu*, *s*, *steps=Params.steps*)

pyuncertainnumber.pba.pbox.**genlogistic**(**args*)

pyuncertainnumber.pba.pbox.**gennorm**(**args*)

pyuncertainnumber.pba.pbox.**genpareto**(**args*)

pyuncertainnumber.pba.pbox.**genexpon**(**args*)

pyuncertainnumber.pba.pbox.**genextreme**(**args*)

pyuncertainnumber.pba.pbox.**gausshyper**(**args*)

pyuncertainnumber.pba.pbox.**gamma**(**args*)

pyuncertainnumber.pba.pbox.**gengamma**(**args*)

pyuncertainnumber.pba.pbox.**genhalflogistic**(**args*)

pyuncertainnumber.pba.pbox.**geninvgauss**(**args*)

pyuncertainnumber.pba.pbox.**gompertz**(**args*)

pyuncertainnumber.pba.pbox.**gumbel_r**(**args*)

pyuncertainnumber.pba.pbox.**gumbel_l**(**args*)

pyuncertainnumber.pba.pbox.**halfcauchy**(**args*)

pyuncertainnumber.pba.pbox.**halflogistic**(**args*)

pyuncertainnumber.pba.pbox.**halfnorm**(**args*)

pyuncertainnumber.pba.pbox.**halfgennorm**(**args*)

pyuncertainnumber.pba.pbox.**hypsecant**(**args*)

pyuncertainnumber.pba.pbox.**invgamma**(**args*)

pyuncertainnumber.pba.pbox.**invgauss**(**args*)

pyuncertainnumber.pba.pbox.**invweibull**(**args*)

pyuncertainnumber.pba.pbox.**irwinhall**(**args*)

pyuncertainnumber.pba.pbox.**jf_skew_t**(**args*)

pyuncertainnumber.pba.pbox.**johnsonsb**(**args*)

pyuncertainnumber.pba.pbox.**johnsonsu**(**args*)

pyuncertainnumber.pba.pbox.**kappa4**(**args*)

pyuncertainnumber.pba.pbox.**kappa3**(**args*)

pyuncertainnumber.pba.pbox.**ksone**(**args*)

pyuncertainnumber.pba.pbox.**kstwo**(**args*)

pyuncertainnumber.pba.pbox.**kstwobign**(**args*)

pyuncertainnumber.pba.pbox.**laplace**(**args*)

pyuncertainnumber.pba.pbox.**laplace_asymmetric**(**args*)

pyuncertainnumber.pba.pbox.**levy**(**args*)

pyuncertainnumber.pba.pbox.**levy_l**(**args*)

pyuncertainnumber.pba.pbox.**levy_stable**(**args*)

pyuncertainnumber.pba.pbox.**logistic**(**args*)

pyuncertainnumber.pba.pbox.**loggamma**(**args*)

pyuncertainnumber.pba.pbox.**loglaplace**(**args*)

pyuncertainnumber.pba.pbox.**loguniform**(**args*)

pyuncertainnumber.pba.pbox.**lomax**(**args*)

pyuncertainnumber.pba.pbox.**maxwell**(**args*)

pyuncertainnumber.pba.pbox.**mielke**(**args*)

pyuncertainnumber.pba.pbox.**moyal**(**args*)

pyuncertainnumber.pba.pbox.**nakagami**(**args*)

pyuncertainnumber.pba.pbox.**ncx2**(**args*)

pyuncertainnumber.pba.pbox.**ncf**(**args*)

pyuncertainnumber.pba.pbox.**nct**(**args*)

pyuncertainnumber.pba.pbox.**norminvgauss**(**args*)

pyuncertainnumber.pba.pbox.**pareto**(**args*)

pyuncertainnumber.pba.pbox.**pearson3**(**args*)

pyuncertainnumber.pba.pbox.**powerlaw**(**args*)

pyuncertainnumber.pba.pbox.**powerlognorm**(**args*)

pyuncertainnumber.pba.pbox.**powernorm**(**args*)

pyuncertainnumber.pba.pbox.**rdist**(**args*)

pyuncertainnumber.pba.pbox.**rayleigh**(**args*)

pyuncertainnumber.pba.pbox.**rel_breitwigner**(**args*)

pyuncertainnumber.pba.pbox.**rice**(**args*)

pyuncertainnumber.pba.pbox.**recipinvgauss**(**args*)

pyuncertainnumber.pba.pbox.**semicircular**(**args*)

pyuncertainnumber.pba.pbox.**skewcauchy**(**args*)

pyuncertainnumber.pba.pbox.**skewnorm**(**args*)

pyuncertainnumber.pba.pbox.**studentized_range**(**args*)

pyuncertainnumber.pba.pbox.**t**(**args*)

pyuncertainnumber.pba.pbox.**trapezoid**(**args*)

pyuncertainnumber.pba.pbox.**triang**(**args*)

pyuncertainnumber.pba.pbox.**truncexpon**(**args*)

pyuncertainnumber.pba.pbox.**truncnorm**(**args*)

pyuncertainnumber.pba.pbox.**truncpareto**(**args*)

pyuncertainnumber.pba.pbox.**truncweibull_min**(**args*)

pyuncertainnumber.pba.pbox.**tukeylambda**(**args*)

pyuncertainnumber.pba.pbox.**uniform_sps**(**args*)

pyuncertainnumber.pba.pbox.**vonmises**(**args*)

pyuncertainnumber.pba.pbox.**vonmises_line**(**args*)

pyuncertainnumber.pba.pbox.**wald**(**args*)

pyuncertainnumber.pba.pbox.**weibull_min**(**args*)

pyuncertainnumber.pba.pbox.**weibull_max**(**args*)

pyuncertainnumber.pba.pbox.**wrapcauchy**(**args*)

pyuncertainnumber.pba.pbox.**trapz**(*a*, *b*, *c*, *d*, *steps=Params.steps*)

pyuncertainnumber.pba.pbox.**truncnorm**(*left*, *right*, *mean=None*, *stddev=None*, *steps=Params.steps*)

pyuncertainnumber.pba.pbox.**uniform**(*a*, *b*, *steps=Params.steps*)

special case of Uniform distribution as Scipy has an unbelivably strange parameterisation than common sense

Parameters

- **a** (-) – (float) lower endpoint
- **b** (-) – (float) upper endpoints

pyuncertainnumber.pba.pbox.**weibull**(**args*, *steps=Params.steps*)

pyuncertainnumber.pba.pbox.**KM**(*k*, *m*, *steps=Params.steps*)

pyuncertainnumber.pba.pbox.**KN**(*k*, *n*, *steps=Params.steps*)

pyuncertainnumber.pba.pbox.**bernoulli**(**args*)

pyuncertainnumber.pba.pbox.**betabinom**(**args*)

pyuncertainnumber.pba.pbox.**betanbinom**(**args*)

pyuncertainnumber.pba.pbox.**binom**(**args*)

pyuncertainnumber.pba.pbox.**boltzmann**(**args*)

pyuncertainnumber.pba.pbox.**dlaplace**(**args*)

pyuncertainnumber.pba.pbox.**geom**(**args*)

pyuncertainnumber.pba.pbox.**hypergeom**(**args*)

pyuncertainnumber.pba.pbox.**logser**(**args*)

pyuncertainnumber.pba.pbox.**nbinom**(**args*)

pyuncertainnumber.pba.pbox.**nchypergeom_fisher**(**args*)

pyuncertainnumber.pba.pbox.**nchypergeom_wallenius**(**args*)

pyuncertainnumber.pba.pbox.**nhypergeom**(**args*)

pyuncertainnumber.pba.pbox.**planck**(**args*)

pyuncertainnumber.pba.pbox.**poisson**(**args*)

pyuncertainnumber.pba.pbox.**randint**(**args*)

pyuncertainnumber.pba.pbox.**skellam**(**args*)

pyuncertainnumber.pba.pbox.**yulesimon**(**args*)

pyuncertainnumber.pba.pbox.**zipf**(**args*)

pyuncertainnumber.pba.pbox.**zipfian**(**args*)

pyuncertainnumber.pba.pbox.**normal**

pyuncertainnumber.pba.pbox.**N**

pyuncertainnumber.pba.pbox.**gaussian**

pyuncertainnumber.pba.pbox.**U**

pyuncertainnumber.pba.pbox.**lognorm**

pyuncertainnumber.pba.pbox.**named_pbox**

pyuncertainnumber.pba.pbox_base

Exceptions

NotIncreasingError	Common base class for all non-exit exceptions.

Classes

Pbox	A probability distribution is a mathematical function that gives the probabilities of occurrence for different possible values of a variable. Probability boxes (p-boxes) represent interval bounds on probability distributions. The left and right quantiles are each stored as a NumPy array containing the percent point function (the inverse of the cumulative distribution function) for *steps* evenly spaced values between 0 and 1. P-boxes can be defined using all the probability distributions that are available through SciPy's statistics library. Naturally, precis probability distributions can be defined by defining a p-box with precise inputs. This means that within probability bounds analysis probability distributions are considered a special case of a p-box with zero width. Distribution-free p-boxes can also be generated when the underlying distribution is unknown but parameters such as the mean, variance or minimum/maximum bounds are known. Such p-boxes make no assumption about the shape of the distribution and instead return bounds expressing all possible distributions that are valid given the known information. Such p-boxes can be constructed making use of Chebyshev, Markov and Cantelli inequalities from probability theory.

Functions

truncate(pbox, min, max)	

Module Contents

exception pyuncertainnumber.pba.pbox_base.**NotIncreasingError**

Bases: Exception

Common base class for all non-exit exceptions.

class pyuncertainnumber.pba.pbox_base.**Pbox**(*left=None*, *right=None*, *steps=None*, *shape=None*, *mean_left=None*, *mean_right=None*, *var_left=None*, *var_right=None*, *interpolation='linear'*)

A probability distribution is a mathematical function that gives the probabilities of occurrence for different possible values of a variable. Probability boxes (p-boxes) represent interval bounds on probability distributions. The left and right quantiles are each stored as a NumPy array containing the percent point function (the inverse of the cumulative distribution function) for *steps* evenly spaced values between 0 and 1. P-boxes can be defined using all the probability distributions that are available through SciPy's statistics library. Naturally, precis probability distributions can be defined by defining a p-box with precise inputs. This means that within probability bounds analysis probability distributions are considered a special case of a p-box with zero width. Distribution-free p-boxes can also be generated when the underlying distribution is unknown but parameters such as the mean, variance or minimum/maximum bounds are known. Such p-boxes make no assumption about the shape of the distribution and instead return bounds expressing all possible distributions that are valid given the known in-

formation. Such p-boxes can be constructed making use of Chebyshev, Markov and Cantelli inequalities from probability theory.

shape = None

__repr__()

__str__

__iter__()

__neg__()

__lt__(*other*)

__rlt__(*other*)

__le__(*other*)

__rle__(*other*)

__gt__(*other*)

__rgt__(*other*)

__ge__(*other*)

__rge__(*other*)

__and__(*other*)

__rand__(*other*)

__or__(*other*)

__ror__(*other*)

__add__(*other*)

__radd__(*other*)

__sub__(*other*)

__rsub__(*other*)

__mul__(*other*)

__rmul__(*other*)

__pow__(*other*)

__rpow__(*other*)

__truediv__(*other*)

__rtruediv__(*other*)

property range

leslie defined range property

property lo

Returns the left-most value in the interval

property hi

Returns the right-most value in the interval

get_range()

get the quantile range of either a pbox or a distribution

_computemoments()

_checkmoments()

cutv(*x*)

get the bounds on the cumulative probability associated with any x-value

cuth(*p=0.5*)

get the bounds on the quantile at any particular probability level

outer_approximate(*n=100*)

outer approximation of a p-box

> **Note**
>
> - *the_interval_list* will have length one less than that of *p_values* (i.e. 100 and 99)

_unary(**args*, *function=lambda x: ...*)

for monotonic unary functions only

exp()

sqrt()

recip()

static check_dependency(*method*)

constant_shape_check()

a helper drop in for define binary ops

steps_check(*other*)

add(*other: Self* | pyuncertainnumber.pba.interval.Interval | *float* | *int*, *method='f'*) → Self

addtion of uncertain numbers with the defined dependency method

pow(*other: Self* | pyuncertainnumber.pba.interval.Interval | *float* | *int*, *method='f'*) → Self

Raises a p-box to the power of other using the defined dependency method

sub(*other*, *method='f'*)

mul(*other*, *method='f'*)

Multiplication of uncertain numbers with the defined dependency method

div(*other*, *method='f'*)

lt(*other*, *method='f'*)

le(*other*, *method='f'*)

gt(*other*, *method='f'*)

ge(*other*, *method='f'*)

min(*other*, *method='f'*)

Returns a new Pbox object that represents the element-wise minimum of two Pboxes.

Parameters

- **other** (-) – Another Pbox object or a numeric value.
- **method** (-) – Calculation method to determine the minimum. Can be one of 'f', 'p', 'o', 'i'.

Returns

Pbox

max(*other*, *method='f'*)

truncate(*a*, *b*, *method='f'*)

Equivalent to self.min(a,method).max(b,method)

env(*other*)

Computes the envelope of two Pboxes.

Parameters: - other: Pbox or numeric value
The other Pbox or numeric value to compute the envelope with.
Returns: - Pbox
The envelope Pbox.
Raises: - ArithmeticError: If both Pboxes have different number of steps.

imp(*other*)

Returns the imposition of self with other pbox

> **Note**
>
> - binary imposition between two pboxes only

logicaland(*other*, *method='f'*)

logicalor(*other*, *method='f'*)

get_interval(**args*) → *pyuncertainnumber.pba.interval.Interval*

get_probability(*val*) → *pyuncertainnumber.pba.interval.Interval*

summary() → str

mean() → *pyuncertainnumber.pba.interval.Interval*

Returns the mean of the pbox

median() → *pyuncertainnumber.pba.interval.Interval*

Returns the median of the distribution

support() → *pyuncertainnumber.pba.interval.Interval*

get_x()

returns the x values for plotting

get_y()

returns the y values for plotting

straddles(*N*, *endpoints=True*) → bool

Parameters

- **N** (*numeric*) – Number to check
- **endpoints** (*bool*) – Whether to include the endpoints within the check

Returns

- *True* – If left $\leq N \leq$ right (Assuming *endpoints=True*)
- *False* – Otherwise

straddles_zero(*endpoints=True*) → bool

Checks whether 0 is within the p-box

show(*figax=None*, *now=True*, *title=''*, *x_axis_label='x'*, ***kwargs*)

legacy plotting function

display(*title=''*, *ax=None*, *style='band'*, *fill_color='lightgray'*, *bound_colors=None*, ***kwargs*)

default plotting function

to_ds_old(*discretisation=Params.steps*)

convert to ds object

> **Note**
>
> - without outer approximation

to_ds(*discretisation=Params.steps*)

convert to ds object

pyuncertainnumber.pba.pbox_base.**truncate**(*pbox*, *min*, *max*)

pyuncertainnumber.pba.pbox_nonparam

Functions

KS_bounds(...)	construct free pbox from sample data by Kolmogorov-Smirnoff confidence bounds
known_constraints(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution free p-box based upon the information given.
min_max(→ pyuncertainnumber.pba.pbox_base.Pbox)	Returns a box shaped Pbox. This is equivalent to an nInterval expressed as a Pbox.
min_max_mean(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the minimum, maximum and mean of the variable
min_mean(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the minimum and mean of the variable
mean_std(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the mean and standard deviation of the variable
mean_var(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the mean and variance of the variable
pos_mean_std(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a positive distribution-free p-box based upon the mean and standard deviation of the variable
min_max_mode(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the minimum, maximum, and mode of the variable
min_max_median(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the minimum, maximum and median of the variable
min_max_median_is_mode(...)	Generates a distribution-free p-box based upon the minimum, maximum and median/mode of the variable when median = mode.
symmetric_mean_std(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a symmetrix distribution-free p-box based upon the mean and standard deviation of the variable
min_max_mean_std(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the minimum, maximum, mean and standard deviation of the variable
min_max_mean_var(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the minimum, maximum, mean and standard deviation of the variable
from_percentiles(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon percentiles of the variable

Module Contents

pyuncertainnumber.pba.pbox_nonparam.**KS_bounds**(*s*, *alpha: float*, *display=True*) → Tuple[*pyuncertainnumber.pba.utils.CDF_bundle*, *pyuncertainnumber.pba.utils.CDF_bundle*]

construct free pbox from sample data by Kolmogorov-Smirnoff confidence bounds

Parameters

- **s** (-) – sample data, precise and imprecise
- **dn** (-) – KS critical value at significance level lpha and sample size N;

pyuncertainnumber.pba.pbox_nonparam.**known_constraints**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *median:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *mode:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *std:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *var:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *cv:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *percentiles:* *dict[*pyuncertainnumber.pba.interval.Interval | *float* | *int]* | *None* = *None*, *debug: bool* = *False*, *steps: int* = *Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution free p-box based upon the information given. This function works by calculating every possible non-parametric p-box that can be generated using the information provided. The returned p-box is the intersection of these p-boxes.

Parameters:

`minimum`: Minimum value of the variable `maximum`: Maximum value of the variable `mean`: Mean value of the variable `median`: Median value of the variable `mode`: Mode value of the variable `std`: Standard deviation of the variable `var`: Variance of the variable `cv`: Coefficient of variation of the variable `percentiles`: Dictionary of percentiles and their values (e.g. {0.1: 1, 0.5: 2, 0.9: nInterval(3,4)}) `steps`: Number of steps to use in the p-box

Error

`ValueError`: If any of the arguments are not consistent with each other. (i.e. if `std` and `var` are both given, but `std != sqrt(var)`)

Returns:

`Pbox`: Imposition of possible p-boxes

pyuncertainnumber.pba.pbox_nonparam.**min_max**(*a:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *b:* pyuncertainnumber.pba.interval.Interval | *float* | *int* = *None*, *steps=Params.steps*, *shape='box'*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Returns a box shaped Pbox. This is equivalent to an nInterval expressed as a Pbox.

Parameters:

`a` : Left side of box `b`: Right side of box

Returns:

`Pbox`

pyuncertainnumber.pba.pbox_nonparam.**min_max_mean**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum, maximum and mean of the variable

Parameters:

`minimum` : minimum value of the variable

`maximum` : maximum value of the variable

`mean` : mean value of the variable

Returns:

`Pbox`

pyuncertainnumber.pba.pbox_nonparam.**min_mean**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps=Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum and mean of the variable

Parameters:

`minimum` : minimum value of the variable

`mean` : mean value of the variable

Returns:

`Pbox`

pyuncertainnumber.pba.pbox_nonparam.**mean_std**(*mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *std:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps=Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the mean and standard deviation of the variable

Parameters:

`mean` : mean of the variable

`std` : standard deviation of the variable

Returns:

`Pbox`

pyuncertainnumber.pba.pbox_nonparam.**mean_var**(*mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *var:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps=Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the mean and variance of the variable

Equivalent to *mean_std(mean,np.sqrt(var))*

Parameters:

`mean` : mean of the variable

`var` : variance of the variable

Returns:

`Pbox`

pyuncertainnumber.pba.pbox_nonparam.**pos_mean_std**(*mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *std:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps=Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a positive distribution-free p-box based upon the mean and standard deviation of the variable

Parameters:

`mean` : mean of the variable

`std` : standard deviation of the variable

Returns:

`Pbox`

pyuncertainnumber.pba.pbox_nonparam.**min_max_mode**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *mode:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum, maximum, and mode of the variable

Parameters:

`minimum` : minimum value of the variable

`maximum` : maximum value of the variable

`mode` : mode value of the variable

Returns:

`Pbox`

pyuncertainnumber.pba.pbox_nonparam.**min_max_median**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *median:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum, maximum and median of the variable

Parameters:

`minimum` : minimum value of the variable

`maximum` : maximum value of the variable

`median` : median value of the variable

Returns:

`Pbox`

pyuncertainnumber.pba.pbox_nonparam.**min_max_median_is_mode**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *m:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum, maximum and median/mode of the variable when median = mode.

Parameters:

`minimum` : minimum value of the variable

`maximum` : maximum value of the variable

`m` : m = median = mode value of the variable

Returns:

`Pbox`

pyuncertainnumber.pba.pbox_nonparam.**symmetric_mean_std**(*mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *std:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a symmetrix distribution-free p-box based upon the mean and standard deviation of the variable

Parameters:

`mean` : mean value of the variable `std` : standard deviation of the variable

Returns

`Pbox`

pyuncertainnumber.pba.pbox_nonparam.**min_max_mean_std**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *std:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum, maximum, mean and standard deviation of the variable

Parameters

`minimum` : minimum value of the variable `maximum` : maximum value of the variable `mean` : mean value of the variable `std` :standard deviation of the variable

Returns

`Pbox`

See also

min_max_mean_var()

pyuncertainnumber.pba.pbox_nonparam.**min_max_mean_var**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *var:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum, maximum, mean and standard deviation of the variable

Parameters

`minimum` : minimum value of the variable `maximum` : maximum value of the variable `mean` : mean value of the variable `var` :variance of the variable

Returns

`Pbox`

Implementation

Equivalent to `min_max_mean_std(minimum,maximum,mean,np.sqrt(var))`

See also

min_max_mean_std()

pyuncertainnumber.pba.pbox_nonparam.**from_percentiles**(*percentiles: dict*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon percentiles of the variable

Parameters

`percentiles` : dictionary of percentiles and their values (e.g. {0: 0, 0.1: 1, 0.5: 2, 0.9: nInterval(3,4), 1:5})

`steps` : number of steps to use in the p-box

Important

The percentiles dictionary is of the form {percentile: value}. Where value can either be a number or an nInterval. If value is a number, the percentile is assumed to be a point percentile. If value is an nInterval, the percentile is assumed to be an interval percentile.

Warning

If no keys for 0 and 1 are given, `-np.inf` and `np.inf` are used respectively. This will result in a p-box that is not bounded and raise a warning.

If the percentiles are not increasing, the percentiles will be intersected. This may not be desired behaviour.

Error

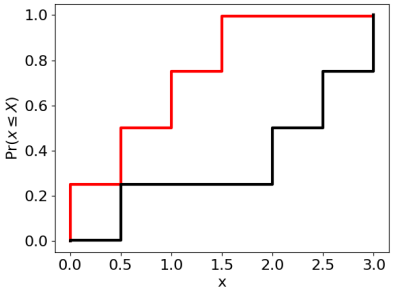
`ValueError`: If any of the percentiles are not between 0 and 1.

Returns

Pbox

Example:

```
pba.from_percentiles(
    {0: 0,
    0.25: 0.5,
    0.5: pba.I(1,2),
    0.75: pba.I(1.5,2.5),
    1: 3}
).show()
```



pyuncertainnumber.pba.utils

Exceptions

NotIncreasingError	Common base class for all non-exit exceptions.

Classes

cdf_bundle
CDF_bundle

Functions

transform_ecdf_bundle(e)	utility to tranform sps.ecdf to cdf_bundle
pl_ecdf_bounding_bundles(b_l, b_r[, alpha, ax, ...])	
sorting(list1, list2)	
weighted_ecdf(s[, w, display])	compute the weighted ecdf from (precise) sample data
reweighting(*masses)	reweight the masses to sum to 1
round()	
uniform_reparameterisation(a, b)	reparameterise the uniform distribution to a, b
find_nearest(array, value)	find the index of the nearest value in the array to the given value
plot_intervals(vec_interval[, ax])	plot the intervals in a vectorised form
_interval_list_to_array(l[, left])	
read_json(file_name)	
check_increasing(arr)	

Module Contents

class pyuncertainnumber.pba.utils.**cdf_bundle**

Bases: `tuple`

quantiles

probabilities

class pyuncertainnumber.pba.utils.**CDF_bundle**

quantiles: numpy.ndarray

probabilities: numpy.ndarray

classmethod from_sps_ecdf(*e*)

utility to tranform sps.ecdf to cdf_bundle

pyuncertainnumber.pba.utils.**transform_ecdf_bundle**(*e*)

utility to tranform sps.ecdf to cdf_bundle

pyuncertainnumber.pba.utils.**pl_ecdf_bounding_bundles**(*b_l:* CDF_bundle, *b_r:* CDF_bundle, *alpha=0.025*, *ax=None*, *legend=True*, *title=None*)

pyuncertainnumber.pba.utils.**sorting**(*list1*, *list2*)

pyuncertainnumber.pba.utils.**weighted_ecdf**(*s*, *w=None*, *display=False*)

compute the weighted ecdf from (precise) sample data

> **Note**

- Sudret eq.1

pyuncertainnumber.pba.utils.**reweighting**(**masses*)

reweight the masses to sum to 1

pyuncertainnumber.pba.utils.**round**()

pyuncertainnumber.pba.utils.**uniform_reparameterisation**(*a*, *b*)

reparameterise the uniform distribution to a, b

pyuncertainnumber.pba.utils.**find_nearest**(*array*, *value*)

find the index of the nearest value in the array to the given value

pyuncertainnumber.pba.utils.**plot_intervals**(*vec_interval: list[*pyuncertainnumber.pba.interval.Interval | pyuncertainnumber.pba.interval.Interval*]*, *ax=None*, ***kwargs*)

plot the intervals in a vectorised form :param vec_interval: vectorised interval objects

pyuncertainnumber.pba.utils.**_interval_list_to_array**(*l*, *left=True*)

pyuncertainnumber.pba.utils.**read_json**(*file_name*)

pyuncertainnumber.pba.utils.**check_increasing**(*arr*)

exception pyuncertainnumber.pba.utils.**NotIncreasingError**

Bases: Exception

Common base class for all non-exit exceptions.

Attributes

normal
N
gaussian
U
lognorm
named_pbox

Classes

nInterval	An interval is an uncertain number for which only the endpoints are known, $x = [a, b]$.
Pbox	A probability distribution is a mathematical function that gives the probabilities of occurrence for different possible values of a variable. Probability boxes (p-boxes) represent interval bounds on probability distributions. The left and right quantiles are each stored as a NumPy array containing the percent point function (the inverse of the cumulative distribution function) for *steps* evenly spaced values between 0 and 1. P-boxes can be defined using all the probability distributions that are available through SciPy's statistics library. Naturally, precis probability distributions can be defined by defining a p-box with precise inputs. This means that within probability bounds analysis probability distributions are considered a special case of a p-box with zero width. Distribution-free p-boxes can also be generated when the underlying distribution is unknown but parameters such as the mean, variance or minimum/maximum bounds are known. Such p-boxes make no assumption about the shape of the distribution and instead return bounds expressing all possible distributions that are valid given the known information. Such p-boxes can be constructed making use of Chebyshev, Markov and Cantelli inequalities from probability theory.
Params	

Functions

wc_interval(bound)	wildcard scalar interval
_get_bounds(dist_family, *args[, steps])	from distribution specification to define the lower and upper bounds of the p-box
_bound_pcdf(dist_family, *args[, steps])	bound the parametric CDF
makePbox(func)	
norm(*args)	
lognormal(mean, var[, steps])	Creates a p-box for the lognormal distribution
alpha(*args)	
anglit(*args)	
argus(*args)	
arcsine(*args)	
beta(*args[, steps])	Beta distribution

continues on next page

Table 4 – continued from previous page

betaprime(*args)
bradford(*args)
burr(*args)
burr12(*args)
cauchy(*args)
chi(*args)
chi2(*args)
cosine(*args)
crystalball(*args)
dgamma(*args)
dweibull(*args)
erlang(*args)
expon(*args)
exponnorm(*args)
exponweib(*args)
exponpow(*args)
f(*args)
fatiguelife(*args)
fisk(*args)
foldcauchy(*args)
foldnorm(mu, s[, steps])
genlogistic(*args)
gennorm(*args)
genpareto(*args)
genexpon(*args)
genextreme(*args)

continues on next page

Table 4 – continued from previous page

gausshyper(*args)
gamma(*args)
gengamma(*args)
genhalflogistic(*args)
geninvgauss(*args)
gompertz(*args)
gumbel_r(*args)
gumbel_l(*args)
halfcauchy(*args)
halflogistic(*args)
halfnorm(*args)
halfgennorm(*args)
hypsecant(*args)
invgamma(*args)
invgauss(*args)
invweibull(*args)
irwinhall(*args)
jf_skew_t(*args)
johnsonsb(*args)
johnsonsu(*args)
kappa4(*args)
kappa3(*args)
ksone(*args)
kstwo(*args)
kstwobign(*args)
laplace(*args)

continues on next page

Table 4 – continued from previous page

laplace_asymmetric(*args)
levy(*args)
levy_l(*args)
levy_stable(*args)
logistic(*args)
loggamma(*args)
loglaplace(*args)
loguniform(*args)
lomax(*args)
maxwell(*args)
mielke(*args)
moyal(*args)
nakagami(*args)
ncx2(*args)
ncf(*args)
nct(*args)
norminvgauss(*args)
pareto(*args)
pearson3(*args)
powerlaw(*args)
powerlognorm(*args)
powernorm(*args)
rdist(*args)
rayleigh(*args)
rel_breitwigner(*args)
rice(*args)

continues on next page

Table 4 – continued from previous page

recipinvgauss(*args)	
semicircular(*args)	
skewcauchy(*args)	
skewnorm(*args)	
studentized_range(*args)	
t(*args)	
trapezoid(*args)	
triang(*args)	
truncexpon(*args)	
truncnorm(left, right[, mean, stddev, steps])	
truncpareto(*args)	
truncweibull_min(*args)	
tukeylambda(*args)	
uniform_sps(*args)	
vonmises(*args)	
vonmises_line(*args)	
wald(*args)	
weibull_min(*args)	
weibull_max(*args)	
wrapcauchy(*args)	
trapz(a, b, c, d[, steps])	
uniform(a, b[, steps])	special case of Uniform distribution as
weibull(*args[, steps])	
KM(k, m[, steps])	
KN(k, n[, steps])	
bernoulli(*args)	

continues on next page

Table 4 – continued from previous page

betabinom(*args)
betanbinom(*args)
binom(*args)
boltzmann(*args)
dlaplace(*args)
geom(*args)
hypergeom(*args)
logser(*args)
nbinom(*args)
nchypergeom_fisher(*args)
nchypergeom_wallenius(*args)
nhypergeom(*args)
planck(*args)
poisson(*args)
randint(*args)
skellam(*args)
yulesimon(*args)
zipf(*args)
zipfian(*args)

Package Contents

class pyuncertainnumber.pba.**nInterval**(*left=None*, *right=None*)

An interval is an uncertain number for which only the endpoints are known, $x = [a, b]$. This is interpreted as x being between a and b but with no more information about the value of x.

Intervals embody epistemic uncertainty within PBA.

Creation

Intervals can be created using either of the following:

```
>>> pba.Interval(0,1)
Interval [0,1]
>>> pba.I(2,3)
Interval [2,3]
```

Tip

The shorthand `I` is an alias for `Interval`

Intervals can also be created from a single value ± half-width:

```
>>> pba.PM(0,1)
Interval [-1,1]
```

By default intervals are displayed as `Interval [a,b]` where `a` and `b` are the left and right endpoints respectively. This can be changed using the **`interval.pm_repr`_** and **`interval.lr_repr`_** functions.

Arithmetic

For two intervals [a,b] and [c,d] the following arithmetic operations are defined:

Addition

$[a, b] + [c, d] = [a + c, b + d]$

Subtraction

$[a, b] - [c, d] = [a - d, b - c]$

Multiplication

$[a, b] * [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$

Division

$[a, b]/[c, d] = [a, b] * \frac{1}{[c,d]} \equiv [\min(a/c, a/d, b/c, b/d), \max(a/c, a/d, b/c, b/d)]$

Alternative arithmetic methods are described in *interval.add*, *interval.sub*, *interval.mul*, *interval.div*.

_left = None

_right = None

property left

property right

__repr__() → str

__str__() → str

__format__(*format_spec: str*) → str

__iter__()

__len__()

__radd__(*left*)

__sub__(*other*)

__rsub__(*other*)

__neg__()

__mul__(*other*)

__rmul__(*other*)

__truediv__(*other*)

__rtruediv__(*other*)

__pow__(*other*)

__rpow__(*left*)

__lt__(*other*)

__rgt__(*other*)

__eq__(*other*)

==

__gt__(*other*)

>

__rlt__(*other*)

__ne__(*other*)

!=

__le__(*other*)

<=

__ge__(*other*)

__bool__()

__abs__()

__contains__(*other*)

add(*other*, *method=None*)

Adds the interval and another object together.

Args:

`other`: The interval or numeric value to be added. This value must be transformable into an Interval object.

Methods:

p - perfect arithmetic $[a, b] + [c, d] = [a + c, b + d]$

o - opposite arithmetic $[a, b] + [c, d] = [a + d, b + c]$

None, i, f - Standard interval arithmetic is used.

Returns:

`Interval`

__add__(*other*)

padd(*other*)

> **Warning**
>
> This method is deprecated. Use add(other, method='p') instead.

oadd(*other*)

> **Warning**
>
> This method is deprecated. Use add(other, method='o') instead.

sub(*other*, *method=None*)

Subtracts other from self.

Args:

`other`: The interval or numeric value to be subracted. This value must be transformable into an Interval object.

Methods:

`p`: perfect arithmetic $a + b = [a.left - b.left, a.right - b.right]$

`o`: opposite arithmetic $a + b = [a.left - b.right, a.right - b.left]$

None, i, f - Standard interval arithmetic is used.

Returns:

`Interval`

psub(*other*)

> **Warning**
>
> Depreciated use self.sub(other, method = 'p') instead

osub(*other*)

> **Warning**
>
> Depreciated use self.sub(other, method = 'o') instead

mul(*other*, *method=None*)

Multiplies self by other.

Args:

`other`: The interval or numeric value to be multiplied. This value must be transformable into an Interval object.

Methods:

`p`: perfect arithmetic $[a, b], [c, d] = [a * c, b * d]$

`o`: opposite arithmetic $[a, b], [c, d] = [a * d, b * c]$

None, i, f - Standard interval arithmetic is used.

Returns:
Interval: The result of the multiplication.

pmul(*other*)

> **Warning**
>
> Depreciated use self.mul(other, method = 'p') instead

omul(*other*)

> **Warning**
>
> Depreciated use self.mul(other, method = 'o') instead

div(*other*, *method=None*)

Divides self by other

If $0 \in other$ it returns a division by zero error

Args:
`other` (Interval or numeric): The interval or numeric value to be multiplied. This value must be transformable into an Interval object.

Methods:
`p`: perfect arithmetic $[a,b],[c,d] = [a * 1/c, b * 1/d]$

`o`: opposite arithmetic $[a,b],[c,d] = [a * 1/d, b * 1/c]$

`None`, `i`, `f` - Standard interval arithmetic is used.

> **Implementation**
>
> ```
> >>> self.add(1/other, method = method)
> ```

> **Error**
>
> If $0 \in [a,b]$ it returns a division by zero error

pdiv(*other*)

> **Warning**
>
> Depreciated use self.div(other, method = 'p') instead

odiv(*other*)

> **Warning**
>
> Depreciated use self.div(other, method = 'o') instead

recip()

Calculates the reciprocle of the interval.

Returns:
`Interval`: Equal to $[1/b, 1/a]$

Example:

```
>>> pba.Interval(2,4).recip()
Interval [0.25, 0.5]
```

> **Error**
>
> If $0 \in [a, b]$ it returns a division by zero error

equiv(*other:* Interval) → bool

Checks whether two intervals are equivalent.

Parameters:
`other`: The interval to check against.

Returns `True` **if**:
`self.left == other.right` and `self.right == other.right`

`False` otherwise.

> **Error**
>
> `TypeError`: If `other` is not an instance of `Interval`

> **See also**
>
> *is_same_as()*

Examples:

```
>>> a = Interval(0,1)
>>> b = Interval(0.5,1.5)
>>> c = I(0,1)
>>> a.equiv(b)
False
>>> a.equiv(c)
True
```

property lo

Returns:
`self.left`

> **Tip**
>
> This function is redundant but exists to match Pbox class for possible internal reasons.

property hi

Returns:
`self.right`

> **Tip**
>
> This function is redundant but exists to match Pbox class for possible internal reasons.

width() → float

Returns:
`float`: The width of the interval, $\text{right} - \text{left}$

Example:

```
>>> pba.Interval(0,3).width()
3
```

halfwidth() → float

Returns:
`float`: The half-width of the interval, $(\text{right} - \text{left})/2$

Example:

```
>>> pba.Interval(0,3).halfwidth()
1.5
```

> **Implementation**
>
> ```
> >>> self.width()/2
> ```

midpoint() → float

Returns the midpoint of the interval

> **Note**
>
> - this serves as the deterministic value representation of the interval, a.k.a. the naked value for an interval

Returns
The midpoint of the interval, $(\text{right} + \text{left})/2$

Return type
`float`

Example:

```
>>> pba.Interval(0,2).midpoint()
1.0
```

`to_logical()`

Turns the interval into a logical interval, this is done by chacking the truth value of the ends of the interval

Returns:

`Logical`: The logical interval

Implementation

```
>>> left = self.left.__bool__()
>>> right = self.right.__bool__()
>>> Logical(left,right)
```

env(*other: list* | Interval) → *Interval*

Calculates the envelope between two intervals

Parameters:

`other` : Interval or list. The interval to envelope with self

Hint

If other is a list then the envelope is calculated between self and each element of the list. In this case the envelope is calculated recursively and pba.envelope() may be more efficient.

Important

If other is a Pbox then `Pbox.env()` is called

See also

`pba.core.envelope`_

`pba.pbox.Pbox.env`_

Returns:

`Interval`: The envelope of self and other

straddles(*N: int* | *float* | Interval, *endpoints: bool = True*) → bool

Parameters:

`N`: Number to check. If N is an interval checks whether the whole interval is within self.

`endpoints`: Whether to include the endpoints within the check

Returns `True` **if**:

left $\leq N \leq$ right (Assuming `endpoints=True`).

For interval values. left $\leq N.left \leq$ right and left $\leq N.right \leq$ right (Assuming `endpoints=True`).

`False` otherwise.

Tip

`N in self` is equivalent to `self.straddles(N)`

straddles_zero(*endpoints=True*)

Checks whether 0 is within the interval

Implementation

Equivalent to `self.straddles(0,endpoints)`

See also

interval.straddles

intersection(*other:* Interval | *list*) → *Interval*

Calculates the intersection between intervals

Parameters:

`other`: The interval to intersect with self. If an interval is not given will try to cast as an interval. If a list is given will calculate the intersection between self and each element of the list.

Returns:

`Interval`: The intersection of self and other. If no intersection is found returns `None`

Example:

```
>>> a = Interval(0,1)
>>> b = Interval(0.5,1.5)
>>> a.intersection(b)
Interval [0.5, 1]
```

exp()

log()

sqrt()

display(*title=''*, *ax=None*, *style='band'*, ***kwargs*)

sample(*seed=None*, *numpy_rng: numpy.random.Generator = None*) → float

Generate a random sample within the interval.

Parameters:

`seed` (int, optional): Seed value for random number generation. Defaults to None.

`numpy_rng` (numpy.random.Generator, optional): Numpy random number generator. Defaults to None.

Returns:

`float`: Random sample within the interval.

Implementation

If `numpy_rng` is given:

```
>>> numpy_rng.uniform(self.left, self.right)
```

Otherwise the following is used:

```
>>> import random
>>> random.seed(seed)
>>> self.left + random.random() * self.width()
```

Examples:

```
>>> pba.Interval(0,1).sample()
0.6160988752201705
>>> pba.I(0,1).sample(seed = 1)
0.13436424411240122
```

If a numpy random number generator is given then it is used instead of the default python random number generator. It has to be initialised first.

```
>>> import numpy as np
>>> rng = np.random.default_rng(seed = 0)
>>> pba.I(0,1).sample(numpy_rng = rng)
0.6369616873214543
```

round()

outward rounding operation for an interval object

classmethod from_midwith(*midpoint: float*, *halfwidth: float*) → *Interval*

Creates an Interval object from a midpoint and half-width.

Parameters

- **midpoint** (-) – The midpoint of the interval.
- **halfwidth** (-) – The half-width of the interval.

Returns

The interval with midpoint and half-width.

Return type

- Interval

Example

```
>>> pba.Interval.from_midwith(0,1)
Interval [-1, 1]
```

class pyuncertainnumber.pba.**Pbox**(*left=None*, *right=None*, *steps=None*, *shape=None*, *mean_left=None*, *mean_right=None*, *var_left=None*, *var_right=None*, *interpolation='linear'*)

A probability distribution is a mathematical function that gives the probabilities of occurrence for different possible values of a variable. Probability boxes (p-boxes) represent interval bounds on probability distributions. The left and right quantiles are each stored as a NumPy array containing the percent point function (the inverse of the cumulative distribution function) for *steps* evenly spaced values between 0 and 1. P-boxes can be defined using all the probability distributions that are available through SciPy's statistics library. Naturally, precis probability distributions can be defined by defining a p-box with precise inputs. This means that within probability bounds analysis probability distributions are considered a special case of a p-box with zero width. Distribution-free p-boxes can also be generated when the underlying distribution is unknown but parameters such as the mean, variance or minimum/maximum bounds are known. Such p-boxes make no assumption about the shape of the

distribution and instead return bounds expressing all possible distributions that are valid given the known information. Such p-boxes can be constructed making use of Chebyshev, Markov and Cantelli inequalities from probability theory.

shape = None

__repr__()

__str__

__iter__()

__neg__()

__lt__(*other*)

__rlt__(*other*)

__le__(*other*)

__rle__(*other*)

__gt__(*other*)

__rgt__(*other*)

__ge__(*other*)

__rge__(*other*)

__and__(*other*)

__rand__(*other*)

__or__(*other*)

__ror__(*other*)

__add__(*other*)

__radd__(*other*)

__sub__(*other*)

__rsub__(*other*)

__mul__(*other*)

__rmul__(*other*)

__pow__(*other*)

__rpow__(*other*)

__truediv__(*other*)

__rtruediv__(*other*)

property range

leslie defined range property

property lo

Returns the left-most value in the interval

property hi

Returns the right-most value in the interval

get_range()

get the quantile range of either a pbox or a distribution

_computemoments()

_checkmoments()

cutv(*x*)

get the bounds on the cumulative probability associated with any x-value

cuth(*p=0.5*)

get the bounds on the quantile at any particular probability level

outer_approximate(*n=100*)

outer approximation of a p-box

> **Note**
>
> - *the_interval_list* will have length one less than that of *p_values* (i.e. 100 and 99)

_unary(**args*, *function=lambda x: ...*)

for monotonic unary functions only

exp()

sqrt()

recip()

static check_dependency(*method*)

constant_shape_check()

a helper drop in for define binary ops

steps_check(*other*)

add(*other: Self* | pyuncertainnumber.pba.interval.Interval | *float* | *int*, *method='f'*) → Self

addtion of uncertain numbers with the defined dependency method

pow(*other: Self* | pyuncertainnumber.pba.interval.Interval | *float* | *int*, *method='f'*) → Self

Raises a p-box to the power of other using the defined dependency method

sub(*other*, *method='f'*)

mul(*other*, *method='f'*)

Multiplication of uncertain numbers with the defined dependency method

div(*other*, *method='f'*)

lt(*other*, *method='f'*)

le(*other*, *method='f'*)

gt(*other*, *method='f'*)

ge(*other*, *method='f'*)

min(*other*, *method='f'*)

Returns a new Pbox object that represents the element-wise minimum of two Pboxes.

Parameters

- **other** (-) – Another Pbox object or a numeric value.
- **method** (-) – Calculation method to determine the minimum. Can be one of 'f', 'p', 'o', 'i'.

Returns

Pbox

max(*other*, *method='f'*)

truncate(*a*, *b*, *method='f'*)

Equivalent to self.min(a,method).max(b,method)

env(*other*)

Computes the envelope of two Pboxes.

Parameters: - other: Pbox or numeric value
The other Pbox or numeric value to compute the envelope with.
Returns: - Pbox
The envelope Pbox.
Raises: - ArithmeticError: If both Pboxes have different number of steps.

imp(*other*)

Returns the imposition of self with other pbox

> **Note**
>
> - binary imposition between two pboxes only

logicaland(*other*, *method='f'*)

logicalor(*other*, *method='f'*)

get_interval(**args*) → *pyuncertainnumber.pba.interval.Interval*

get_probability(*val*) → *pyuncertainnumber.pba.interval.Interval*

summary() → str

mean() → *pyuncertainnumber.pba.interval.Interval*

Returns the mean of the pbox

median() → *pyuncertainnumber.pba.interval.Interval*

Returns the median of the distribution

support() → *pyuncertainnumber.pba.interval.Interval*

get_x()

returns the x values for plotting

get_y()

returns the y values for plotting

straddles(*N*, *endpoints=True*) → bool

Parameters

- **N** (*numeric*) – Number to check
- **endpoints** (*bool*) – Whether to include the endpoints within the check

Returns

- *True* – If left $\leq N \leq$ right (Assuming *endpoints=True*)
- *False* – Otherwise

straddles_zero(*endpoints=True*) → bool

Checks whether 0 is within the p-box

show(*figax=None*, *now=True*, *title=''*, *x_axis_label='x'*, ***kwargs*)

legacy plotting function

display(*title=''*, *ax=None*, *style='band'*, *fill_color='lightgray'*, *bound_colors=None*, ***kwargs*)

default plotting function

to_ds_old(*discretisation=Params.steps*)

convert to ds object

> **Note**
>
> - without outer approximation

to_ds(*discretisation=Params.steps*)

convert to ds object

class pyuncertainnumber.pba.**Params**

hedge_cofficients

steps = 200

many = 2000

p_values

p_lboundary = 0.0001

p_hboundary = 0.9999

scott_hedged_interpretation

user_hedged_interpretation

result_path = './results/'

hw = 0.5

pyuncertainnumber.pba.**wc_interval**(*bound*)

wildcard scalar interval

pyuncertainnumber.pba.**_get_bounds**(*dist_family*, **args*, *steps=Params.steps*)

from distribution specification to define the lower and upper bounds of the p-box

Parameters

dist_family (-) – (str) the name of the distribution

pyuncertainnumber.pba.**_bound_pcdf**(*dist_family*, **args*, *steps=Params.steps*)

bound the parametric CDF

Note

- only support fully bounded parameters

pyuncertainnumber.pba.**makePbox**(*func*)

pyuncertainnumber.pba.**norm**(**args*)

pyuncertainnumber.pba.**lognormal**(*mean*, *var*, *steps=Params.steps*)

Creates a p-box for the lognormal distribution

Note: the parameters used are the mean and variance of the lognormal distribution not the mean and variance of the underlying normal See: *[1]<https://en.wikipedia.org/wiki/Log-normal_distribution#Generation_and_parameters>* *[2]<https://stackoverflow.com/questions/51906063/distribution-mean-and-standard-deviation-using-scipy-stats>*

Parameters

- **mean** – mean of the lognormal distribution
- **var** – variance of the lognormal distribution

Return type

Pbox

pyuncertainnumber.pba.**alpha**(**args*)

pyuncertainnumber.pba.**anglit**(**args*)

pyuncertainnumber.pba.**argus**(**args*)

pyuncertainnumber.pba.**arcsine**(**args*)

pyuncertainnumber.pba.**beta**(**args*, *steps=Params.steps*)

Beta distribution

pyuncertainnumber.pba.**betaprime**(**args*)

pyuncertainnumber.pba.**bradford**(**args*)

pyuncertainnumber.pba.**burr**(**args*)

pyuncertainnumber.pba.**burr12**(**args*)

pyuncertainnumber.pba.**cauchy**(**args*)

pyuncertainnumber.pba.**chi**(**args*)

pyuncertainnumber.pba.**chi2**(**args*)

pyuncertainnumber.pba.**cosine**(**args*)

pyuncertainnumber.pba.**crystalball**(**args*)

pyuncertainnumber.pba.**dgamma**(**args*)

pyuncertainnumber.pba.**dweibull**(**args*)

pyuncertainnumber.pba.**erlang**(**args*)

pyuncertainnumber.pba.**expon**(**args*)

pyuncertainnumber.pba.**exponnorm**(**args*)

pyuncertainnumber.pba.**exponweib**(**args*)

pyuncertainnumber.pba.**exponpow**(**args*)

pyuncertainnumber.pba.**f**(**args*)

pyuncertainnumber.pba.**fatiguelife**(**args*)

pyuncertainnumber.pba.**fisk**(**args*)

pyuncertainnumber.pba.**foldcauchy**(**args*)

pyuncertainnumber.pba.**foldnorm**(*mu*, *s*, *steps=Params.steps*)

pyuncertainnumber.pba.**genlogistic**(**args*)

pyuncertainnumber.pba.**gennorm**(**args*)

pyuncertainnumber.pba.**genpareto**(**args*)

pyuncertainnumber.pba.**genexpon**(**args*)

pyuncertainnumber.pba.**genextreme**(**args*)

pyuncertainnumber.pba.**gausshyper**(**args*)

pyuncertainnumber.pba.**gamma**(**args*)

pyuncertainnumber.pba.**gengamma**(**args*)

pyuncertainnumber.pba.**genhalflogistic**(**args*)

pyuncertainnumber.pba.**geninvgauss**(**args*)

pyuncertainnumber.pba.**gompertz**(**args*)

pyuncertainnumber.pba.**gumbel_r**(**args*)

pyuncertainnumber.pba.**gumbel_l**(**args*)

pyuncertainnumber.pba.**halfcauchy**(**args*)

pyuncertainnumber.pba.**halflogistic**(**args*)

pyuncertainnumber.pba.**halfnorm**(**args*)

pyuncertainnumber.pba.**halfgennorm**(**args*)

pyuncertainnumber.pba.**hypsecant**(**args*)

pyuncertainnumber.pba.**invgamma**(**args*)

pyuncertainnumber.pba.**invgauss**(**args*)

pyuncertainnumber.pba.**invweibull**(**args*)

pyuncertainnumber.pba.**irwinhall**(**args*)

pyuncertainnumber.pba.**jf_skew_t**(**args*)

pyuncertainnumber.pba.**johnsonsb**(**args*)

pyuncertainnumber.pba.**johnsonsu**(**args*)

pyuncertainnumber.pba.**kappa4**(**args*)

pyuncertainnumber.pba.**kappa3**(**args*)

pyuncertainnumber.pba.**ksone**(**args*)

pyuncertainnumber.pba.**kstwo**(**args*)

pyuncertainnumber.pba.**kstwobign**(**args*)

pyuncertainnumber.pba.**laplace**(**args*)

pyuncertainnumber.pba.**laplace_asymmetric**(**args*)

pyuncertainnumber.pba.**levy**(**args*)

pyuncertainnumber.pba.**levy_l**(**args*)

pyuncertainnumber.pba.**levy_stable**(**args*)

pyuncertainnumber.pba.**logistic**(**args*)

pyuncertainnumber.pba.**loggamma**(**args*)

pyuncertainnumber.pba.**loglaplace**(**args*)

pyuncertainnumber.pba.**loguniform**(**args*)

pyuncertainnumber.pba.**lomax**(**args*)

pyuncertainnumber.pba.**maxwell**(**args*)

pyuncertainnumber.pba.**mielke**(**args*)

pyuncertainnumber.pba.**moyal**(**args*)

pyuncertainnumber.pba.**nakagami**(**args*)

pyuncertainnumber.pba.**ncx2**(**args*)

pyuncertainnumber.pba.**ncf**(**args*)

pyuncertainnumber.pba.**nct**(**args*)

pyuncertainnumber.pba.**norminvgauss**(**args*)

pyuncertainnumber.pba.**pareto**(**args*)

pyuncertainnumber.pba.**pearson3**(**args*)

pyuncertainnumber.pba.**powerlaw**(**args*)

pyuncertainnumber.pba.**powerlognorm**(**args*)

pyuncertainnumber.pba.**powernorm**(**args*)

pyuncertainnumber.pba.**rdist**(**args*)

pyuncertainnumber.pba.**rayleigh**(**args*)

pyuncertainnumber.pba.**rel_breitwigner**(**args*)

pyuncertainnumber.pba.**rice**(**args*)

pyuncertainnumber.pba.**recipinvgauss**(**args*)

pyuncertainnumber.pba.**semicircular**(**args*)

pyuncertainnumber.pba.**skewcauchy**(**args*)

pyuncertainnumber.pba.**skewnorm**(**args*)

pyuncertainnumber.pba.**studentized_range**(**args*)

pyuncertainnumber.pba.**t**(**args*)

pyuncertainnumber.pba.**trapezoid**(**args*)

pyuncertainnumber.pba.**triang**(**args*)

pyuncertainnumber.pba.**truncexpon**(**args*)

pyuncertainnumber.pba.**truncnorm**(*left*, *right*, *mean=None*, *stddev=None*, *steps=Params.steps*)

pyuncertainnumber.pba.**truncpareto**(**args*)

pyuncertainnumber.pba.**truncweibull_min**(**args*)

pyuncertainnumber.pba.**tukeylambda**(**args*)

pyuncertainnumber.pba.**uniform_sps**(**args*)

pyuncertainnumber.pba.**vonmises**(**args*)

pyuncertainnumber.pba.**vonmises_line**(**args*)

pyuncertainnumber.pba.**wald**(**args*)

pyuncertainnumber.pba.**weibull_min**(**args*)

pyuncertainnumber.pba.**weibull_max**(**args*)

pyuncertainnumber.pba.**wrapcauchy**(**args*)

pyuncertainnumber.pba.**trapz**(*a*, *b*, *c*, *d*, *steps=Params.steps*)

pyuncertainnumber.pba.**uniform**(*a*, *b*, *steps=Params.steps*)

special case of Uniform distribution as Scipy has an unbelivably strange parameterisation than common sense

Parameters

- **a** (-) – (float) lower endpoint
- **b** (-) – (float) upper endpoints

pyuncertainnumber.pba.**weibull**(**args*, *steps=Params.steps*)

pyuncertainnumber.pba.**KM**(*k*, *m*, *steps=Params.steps*)

pyuncertainnumber.pba.**KN**(*k*, *n*, *steps=Params.steps*)

pyuncertainnumber.pba.**bernoulli**(**args*)

pyuncertainnumber.pba.**betabinom**(**args*)

pyuncertainnumber.pba.**betanbinom**(**args*)

pyuncertainnumber.pba.**binom**(**args*)

pyuncertainnumber.pba.**boltzmann**(**args*)

pyuncertainnumber.pba.**dlaplace**(**args*)

pyuncertainnumber.pba.**geom**(**args*)

pyuncertainnumber.pba.**hypergeom**(**args*)

pyuncertainnumber.pba.**logser**(**args*)

pyuncertainnumber.pba.**nbinom**(**args*)

pyuncertainnumber.pba.**nchypergeom_fisher**(**args*)

pyuncertainnumber.pba.**nchypergeom_wallenius**(**args*)

pyuncertainnumber.pba.**nhypergeom**(**args*)

pyuncertainnumber.pba.**planck**(**args*)

pyuncertainnumber.pba.**poisson**(**args*)

pyuncertainnumber.pba.**randint**(**args*)

pyuncertainnumber.pba.**skellam**(**args*)

pyuncertainnumber.pba.**yulesimon**(**args*)

pyuncertainnumber.pba.**zipf**(**args*)

pyuncertainnumber.pba.**zipfian**(**args*)

pyuncertainnumber.pba.**normal**

pyuncertainnumber.pba.**N**

pyuncertainnumber.pba.**gaussian**

pyuncertainnumber.pba.**U**

pyuncertainnumber.pba.**lognorm**

pyuncertainnumber.pba.**named_pbox**

pyuncertainnumber.propagation

Submodules

pyuncertainnumber.propagation.performance_func

Functions

cb_func(x)	Calculates deflection and stress for a cantilever beam.
cb_deflection(beam_length, I, F, E)	compute the deflection in the cantilever beam example
cb_deflection(beam_length, I, F, E)	compute the deflection in the cantilever beam example
cb_stress(y, beam_length, I, F)	to compute bending stress in the cantilever beam example

Module Contents

pyuncertainnumber.propagation.performance_func.**cb_func**(*x*)

Calculates deflection and stress for a cantilever beam.

Parameters

x (*np.array*) – Array of input parameters: x[0]: Distance from the neutral axis to the point of interest (m) x[1]: Length of the beam (m) x[2]: Second moment of area (mm^4) x[3]: Applied force (N) x[4]: Young's modulus (MPa)

Returns

np.array([deflection (m), stress (MPa)])
Returns np.array([np.nan, np.nan]) if calculation error occurs.

pyuncertainnumber.propagation.performance_func.**cb_deflection**(*x*)

Calculates deflection and stress for a cantilever beam.

Parameters

x (*np.array*) – Array of input parameters: x[0]: Length of the beam (m) x[1]: Second moment of area (mm^4) x[2]: Applied force (N) x[3]: Young's modulus (MPa)

Returns

deflection (m)
Returns np.nan if calculation error occurs.

Return type
float

pyuncertainnumber.propagation.performance_func.**cb_deflection**(*beam_length*, *I*, *F*, *E*)

compute the deflection in the cantilever beam example

TODO add typing for UncertainNumber :param beam_length: Length of the beam (m) :type beam_length: UncertainNumber :param I: Second moment of area (mm^4) :param F: Applied force (N) :param E: Young's modulus (MPa)

Returns

deflection (m)
Returns np.nan if calculation error occurs.

Return type
float

pyuncertainnumber.propagation.performance_func.**cb_stress**(*y*, *beam_length*, *I*, *F*)

to compute bending stress in the cantilever beam example

pyuncertainnumber.propagation.uncertaintyPropagation

Functions

aleatory_propagation([vars, results, fun, n_sam, ...])	This function propagates aleatory uncertainty through a given function (*fun*) using either Monte Carlo or Latin Hypercube sampling, considering the aleatory uncertainty represented by a list of *UncertainNumber* objects (*vars*).
mixed_propagation(vars[, fun, results, method, ...])	Performs mixed uncertainty propagation through a given function. This function handles uncertainty propagation when there's a mix of
epistemic_propagation(vars, fun[, results, n_sub, ...])	Performs epistemic uncertainty propagation through a given function. This function implements various methods for propagating epistemic uncertainty,
Propagation(vars, fun[, results, n_sub, n_sam, x0, ...])	Performs uncertainty propagation through a given function with uncertain inputs. This function automatically selects and executes an appropriate uncertainty propagation method based on the types of uncertainty in the input variables. It supports interval analysis, probabilistic methods, and mixed uncertainty propagation.
plotPbox(xL, xR[, p])	Plots a p-box (probability box) using matplotlib.
main()	implementation of any method for epistemic uncertainty on the cantilever beam example

Module Contents

pyuncertainnumber.propagation.uncertaintyPropagation.**aleatory_propagation**(*vars: list = None*, *results:* pyuncertainnumber.propagation.utils.Propagation_resu *= None*, *fun: Callable = None*, *n_sam: int = 500*, *method: str = 'monte_carlo'*, *save_raw_data='no'*, ***, *base_path=np.nan*, ***kwargs*)

This function propagates aleatory uncertainty through a given function (*fun*) using either Monte Carlo or Latin Hypercube sampling, considering the aleatory uncertainty represented by a list of *UncertainNumber* objects (*vars*). :param - vars: A list of UncertainNumber objects, each representing an input variable with its associated uncertainty.

Parameters

- **fun** (-) – The function to propagate uncertainty through.

- **n_sam** (-) – The number of samples to generate. Default is 500.
- **method** (-) – The sampling method ('monte_carlo' or 'latin_hypercube'). Defaults to 'monte_carlo'.
- **save_raw_data** (-) – Whether to save raw data ('yes' or 'no'). Defaults to 'no'.
- **base_path** (-) – Path for saving results (if save_raw_data is 'yes'). Defaults to np.nan.
- ****kwargs** (-) – Additional keyword arguments to be passed to the UncertainNumber constructor.

signature:
aleatory_propagation(x:np.ndarray, f:Callable, n:int, ...) -> Propagation_results

Note

- If the *f* function returns multiple outputs, the *all_output* array will be 2-dimensional y and x for all x samples.

Returns

A *Propagation_results* object containing:

- **'un': A list of UncertainNumber objects, each representing** the output(s) of the function.
- **'raw_data': A dictionary containing raw data (if**

  save_raw_data is 'yes'):

  - 'x': All generated input samples.
  - **'f': Corresponding output values for each** input sample.

Return type
Propagation_results

Raises
ValueError – For invalid method, save_raw_data, or missing arguments.

pyuncertainnumber.propagation.uncertaintyPropagation.**mixed_propagation**(*vars: list*, *fun: Callable = None*, *results:* pyuncertainnumber.propagation.utils.Propagation_results *= None*, *method='second_order_extremepoints'*, *n_disc: int | numpy.ndarray = 10*, *condensation: int = None*, *tOp: float | numpy.ndarray = 0.999*, *bOt: float | numpy.ndarray = 0.001*, *save_raw_data='no'*, ***, *base_path=np.nan*, ***kwargs*)

Performs mixed uncertainty propagation through a given function. This function handles uncertainty propagation when there's a mix of

aleatory and epistemic uncertainty in the input variables.

Parameters

- **vars** (-) – A list of uncertain variables, which can be a mix of different uncertainty types (e.g., intervals, distributions).
- **fun** (-) – The function to propagate uncertainty through.
- **results** (-) – An object to store propagation results. Defaults to None, in which case a new *Propagation_results* object is created.
- **method** (-) – The mixed uncertainty propagation method. Can be one of: 'second_order_endpoints', 'second_order_vertex', 'second_order_extremepoints', 'first_order_extremepoints'. Defaults to 'second_order_extremepoints'.
- **n_disc** (-) – Number of discretization points for interval variables. Defaults to 10.
- **condensation** (-) – Parameter for reducing the complexity of the output uncertainty representation. Defaults to None.
- **tOp** (-) – Upper threshold or bound used in some methods. Defaults to 0.999.
- **bOt** (-) – Lower threshold or bound used in some methods. Defaults to 0.001.
- **save_raw_data** (-) – Whether to save raw data ('yes' or 'no'). Defaults to 'no'.
- **base_path** (-) – Path for saving results (if save_raw_data is 'yes'). Defaults to np.nan.
- ****kwargs** (-) – Additional keyword arguments passed to the underlying propagation methods.

signature:

mixed_propagation(vars: list, fun: Callable, results: Propagation_results = None, ...) -> Propagation_results

Notes

- It can be used if each uncertain number is exrpessed in terms of precise distributions.

Returns

A *Propagation_results* object containing the results of

the mixed uncertainty propagation. The format of the results depends on the chosen *method*.

Return type

Propagation_results

Raises

ValueError – For invalid *method* or *save_raw_data*.

Examples

```
>>> a = mixed_propagation(vars= [y, L, I, F, E],
                    fun= cantilever_beam_func,
                    method= 'second_order_extremepoints',
                    n_disc=8,
                    #save_raw_data= "no"#,
                    save_raw_data= "yes",
                    base_path= base_path
                )
```

pyuncertainnumber.propagation.uncertaintyPropagation.**epistemic_propagation**(*vars*, *fun*, *results: pyuncertainnumber.propagation.utils.Propagation_res = None*, *n_sub: numpy.integer = None*, *n_sam: numpy.integer = None*, *x0: numpy.ndarray = None*, *method: str = None*, *save_raw_data='no'*, ***, *base_path=np.nan*, *tol_loc: numpy.ndarray = None*, *options_loc: dict = None*, *method_loc='Nelder-Mead'*, *pop_size=1000*, *n_gen=100*, *tol=0.001*, *n_gen_last=10*, *algorithm_type='NSGA2'*, ***kwargs*)

Performs epistemic uncertainty propagation through a given function. This function implements various methods for propagating epistemic uncertainty,

typically represented as intervals.

Parameters

- **vars** (-) – A list of *UncertainNumber* objects representing the input variables with their associated interval uncertainty.
- **fun** (-) – The function to propagate uncertainty through.
- **results** (-) – An object to store propagation results. Defaults to None, in which case a new *Propagation_results* object is created.
- **n_sub** (-) – Number of subintervals for subinterval methods. Defaults to None.
- **n_sam** (-) – Number of samples for sampling-based methods. Defaults to None.
- **x0** (-) – Initial guess for local optimization methods. Defaults to None.
- **method** (-) – The uncertainty propagation method to use. Defaults to “endpoint”.
- **save_raw_data** (-) – Whether to save raw data (‘yes’ or ‘no’). Defaults to “no”.
- **base_path** (-) – Path for saving results (if save_raw_data is ‘yes’). Defaults to np.nan.
- **tol_loc** (-) – Tolerance for local optimization. Defaults to None.
- **options_loc** (-) – Options for local optimization. Defaults to None.
- **method_loc** (-) – Method for local optimization. Defaults to ‘Nelder-Mead’.
- **pop_size** (-) – Population size for genetic algorithms. Defaults to 1000.

- **n_gen** (-) – Number of generations for genetic algorithms. Defaults to 100.
- **tol** (-) – Tolerance for genetic algorithms. Defaults to 1e-3.
- **n_gen_last** (-) – Number of last generations for genetic algorithms. Defaults to 10.
- **algorithm_type** (-) – Type of genetic algorithm. Defaults to 'NSGA2'.
- ****kwargs** (-) – Additional keyword arguments passed to the *UncertainNumber* constructor.

signature:

epistemic_propagation(vars: list, fun: Callable, results: Propagation_results = None, ...) -> Propagation_results

Notes

- It supports a wide range of techniques, including:
  1. **Interval-based methods:**
     - ***endpoints* or *vertex*: Calculates the function output at the endpoints** or vertices of the input intervals.
     - ***extremepoints*: Considers all possible combinations of interval endpoints** to find the extreme values of the output.
     - ***subinterval* or *subinterval_reconstitution*: Divides the input intervals** into subintervals and performs propagation on each subinterval.
  2. **Sampling-based methods:**
     - ***monte_carlo*, *latin_hypercube*: Uses Monte Carlo or Latin Hypercube** sampling within the input intervals.
     - ***monte_carlo_endpoints*, *latin_hypercube_endpoints*: Combines sampling with** evaluation at interval endpoints.
     - ***cauchy*, *endpoint_cauchy*, *endpoints_cauchy*: Uses Cauchy deviates for** sampling.
  3. **Optimization-based methods:**
     - ***local_optimization* or *local_optimisation*: Uses local optimization** algorithms to find the minimum or maximum output values.
     - ***genetic_optimisation* or *genetic_optimization*: Uses genetic algorithms for** global optimization.

Returns

A *Propagation_results* object containing the results of the epistemic uncertainty propagation. The format of the results depends on the chosen *method*.

Return type

- Propagation_results

Raises

- **- ValueError** – For invalid *method*, *save_raw_data*, or missing arguments.
- **- TypeError** – If *fun* is not callable for optimization methods.

Example

```
>>> a = epistemic_propagation(vars= [ y, L, I, F, E],
                          fun= cantilever_beam_func,
                          method= 'extremepoints',
                          n_disc=8,
                          save_raw_data= "no"
                      )
```

pyuncertainnumber.propagation.uncertaintyPropagation.**Propagation**(*vars: list*, *fun: Callable*, *results:* pyuncertainnumber.propagation.utils.Propagation_results *= None*, *n_sub: numpy.integer = 3*, *n_sam: numpy.integer = 500*, *x0: numpy.ndarray = None*, *method=None*, *n_disc: int | numpy.ndarray = 10*, *condensation: int = None*, *tOp: float | numpy.ndarray = 0.999*, *bOt: float | numpy.ndarray = 0.001*, *save_raw_data='no'*, ***, *base_path=np.nan*, *tol_loc: numpy.ndarray = None*, *options_loc: dict = None*, *method_loc='Nelder-Mead'*, *pop_size=1000*, *n_gen=100*, *tol=0.001*, *n_gen_last=10*, *algorithm_type='NSGA2'*, ***kwargs*)

Performs uncertainty propagation through a given function with uncertain inputs. This function automatically selects and executes an appropriate uncertainty propagation method based on the types of uncertainty in the input variables. It supports interval analysis, probabilistic methods, and mixed uncertainty propagation.

Parameters

- **vars** (-) – A list of uncertain variables.
- **fun** (-) – The function through which to propagate uncertainty.
- **results** (-) – An object to store propagation results. Defaults to None, in which case a new *Propagation_results* object is created.
- **n_sub** (-) – Number of subintervals for interval-based methods. Defaults to 3.
- **n_sam** (-) – Number of samples for Monte Carlo simulation. Defaults to 500.
- **x0** (-) – Initial guess for optimization-based methods. Defaults to None.
- **method** (-) – Specifies the uncertainty propagation method. Defaults to None, which triggers automatic selection.
- **n_disc** (-) – Number of discretization points. Defaults to 10.
- **condensation** (-) – Parameter for reducing output complexity. Defaults to None.
- **tOp** (-) – Upper threshold or bound. Defaults to 0.999.
- **bOt** (-) – Lower threshold or bound. Defaults to 0.001.
- **save_raw_data** (-) – Whether to save intermediate results (‘yes’ or ‘no’). Defaults to ‘no’.

- **base_path** (-) – Path for saving data. Defaults to np.nan.
- **tol_loc** (-) – Tolerance for local optimization. Defaults to None.
- **options_loc** (-) – Options for local optimization. Defaults to None.
- **method_loc** (-) – Method for local optimization. Defaults to 'Nelder-Mead'.
- **pop_size** (-) – Population size for genetic algorithms. Defaults to 1000.
- **n_gen** (-) – Number of generations for genetic algorithms. Defaults to 100.
- **tol** (-) – Tolerance for genetic algorithms. Defaults to 1e-3.
- **n_gen_last** (-) – Number of last generations for genetic algorithms. Defaults to 10.
- **algorithm_type** (-) – Type of genetic algorithm. Defaults to 'NSGA2'.
- ****kwargs** – Additional keyword arguments passed to the underlying propagation methods.

signature:

- Propagation(vars: list, fun: Callable, results: Propagation_results = None, ...) -> Propagation_results

Returns

A *Propagation_results* object including:

- **'un': A list of UncertainNumber objects, each representing** the output(s) of the function.
- 'raw_data': depending on the method selected.

Return type

- Propagation_results

Example

```
>>> a = Propagation(vars= [ y, L, I, F, E],
                    fun= cantilever_beam_func,
                    method= 'extremepoints',
                    n_disc=8,
                    save_raw_data= "no"
                )
```

pyuncertainnumber.propagation.uncertaintyPropagation.**plotPbox**(*xL*, *xR*, *p=None*)

Plots a p-box (probability box) using matplotlib.

Parameters

- **xL** (*np.ndarray*) – A 1D NumPy array of lower bounds.
- **xR** (*np.ndarray*) – A 1D NumPy array of upper bounds.
- **p** (*np.ndarray, optional*) – A 1D NumPy array of probabilities corresponding to the intervals. Defaults to None, which generates equally spaced probabilities.
- **color** (*str, optional*) – The color of the plot. Defaults to 'k' (black).

pyuncertainnumber.propagation.uncertaintyPropagation.**main**()

implementation of any method for epistemic uncertainty on the cantilever beam example

pyuncertainnumber.propagation.utils

Classes

Propagation_results	Stores the results of uncertainty propagation with multiple outputs, sharing raw_data x and f.

Functions

header_results(all_output, all_input[, method])	Determine generic header for output and input.
post_processing(all_input[, all_output, method, res_path])	Post-processes the results of an uncertainty propagation (UP) method.
create_folder(base_path, method)	Creates a folder named after the called UP method where the results files are stored
create_csv(res_path, filename, data)	Creates a .csv file and sotres it in a pre-specified folder with results generated by a UP method
save_results(data, method, res_path[, fun])	
condense_bounds(bounds, N)	Condenses lower and upper bounds of a probability distribution to a specified size.

Module Contents

class pyuncertainnumber.propagation.utils.**Propagation_results**(*un: numpy.ndarray = None*, *raw_data: dict = None*)

Stores the results of uncertainty propagation with multiple outputs, sharing raw_data x and f.

Parameters

- **un** (*np.ndarray*) – np.array of UncertainNumber objects (one for each output).
- **raw_data** (*dict*) – Dictionary containing raw data shared across outputs: x (np.ndarray): Input values. f (np.ndarray): Output values. min (np.ndarray): Array of dictionaries, one for each output,

  containing 'x', 'f' for the minimum of that output.

  max (np.ndarray): Array of dictionaries, one for each output,
  containing 'x', 'f' for the maximum of that output.

  bounds (np.ndarray): 2D array of lower and upper bounds for each output.

Notes

- use *foo.un* to access the UncertainNumber objects.

add_raw_data(*x=None*, *f=None*, *K=None*, *sign_x: numpy.ndarray = None*)

Adds raw data to the results.

summary()

Prints the results in a formatted way, handling None values and multiple outputs.

pyuncertainnumber.propagation.utils.**header_results**(*all_output*, *all_input*, *method=None*)

Determine generic header for output and input.

Parameters

- **all_output** (*np.ndarray*) – A NumPy array containing the output values.
- **all_input** (*np.ndarray*) – A NumPy array containing the input values.

Returns
A list of strings representing the header for the combined DataFrame.

Return type
list

pyuncertainnumber.propagation.utils.**post_processing**(*all_input: numpy.ndarray*, *all_output: numpy.ndarray = None*, *method=None*, *res_path=None*)

Post-processes the results of an uncertainty propagation (UP) method.

This function takes the input and output values from a UP method, combines them into a pandas DataFrame, and optionally saves the raw data to a CSV file. It also checks for NaN values in the output and logs them with their corresponding input values if found. If all_output is None, it creates a DataFrame with only the input data.

Parameters

- **all_input** (*np.ndarray*) – A NumPy array containing the input values used in the UP method.
- **all_output** (*np.ndarray, optional*) – A NumPy array containing the corresponding output values from the UP method. Defaults to None.
- **res_path** (*str, optional*) – The path to the directory where the results will be saved. Defaults to None.

Returns

A pandas DataFrame containing the combined output and input data
(if all_output is provided). If all_output is None, it returns a DataFrame with only the input data.

Return type
pandas.DataFrame

pyuncertainnumber.propagation.utils.**create_folder**(*base_path*, *method*)

Creates a folder named after the called UP method where the results files are stored

Parameters

- **base_path** (-) – The base path
- **method** (-) – the name of the called method

signature:
create_folder(base_path: string, method: string) -> path.folder

Note

- the augument *base_path* will specify the location of the created results folder.
- the argument *method* will provide the name for the results folder.

Returns

- A folder in a prespecified path

Example

base_path = “C:/Users/DAWS2_code/UP” method = “vertex” y = create_folder(base_path, method)

pyuncertainnumber.propagation.utils.**create_csv**(*res_path*, *filename*, *data*)

Creates a .csv file and sotres it in a pre-specified folder with results generated by a UP method

Parameters

- **res_path** (-) – A folder in a prespecified path named after the called UP method
- **filename** (-) – the name of the file
- **data** (-) – a pandas.dataframe with results from UP method

signature:

create_csv(res_path = path, filename = filename, data = pandas.dataframe) -> path.filename

Note

- the augument *res_path* will specify the folder where the .csv file will be created.
- argument *file* will provide the name of hte .csv file.
- argument *data* will provide data in terms of pandas.dataframe.

Returns

- A .csv file in a prespecified folder

Example

base_path = “C:/Users/DAWS2_code/UP/vertex” filename = ‘min_max_values’

df = pd.DataFrame(

{“Name”

[[“y0”, “y0”],] “fun” : [“min”,”max”] “y0” : [4, 6]}, index = [1, 2, 3])

header = [‘Name’, ‘fun’, ‘values’] y = create_csv(res_path, filename, df)

pyuncertainnumber.propagation.utils.**save_results**(*data*, *method*, *res_path*, *fun=None*)

pyuncertainnumber.propagation.utils.**condense_bounds**(*bounds*, *N*)

Condenses lower and upper bounds of a probability distribution to a specified size.

Parameters

- **bounds** – A NumPy array of shape (num_outputs, 2, num_points) representing the lower and upper bounds of a probability distribution for potentially multiple outputs. The first dimension corresponds to different outputs of the function, the second dimension corresponds to lower and upper bounds (0 for lower, 1 for upper), and the third dimension corresponds to the original discretization points.
- **N** – The desired size of the condensed arrays.

Returns

A NumPy array of shape (num_outputs, 2, N) containing the condensed lower and upper bounds.

6.7.2 Attributes

normal	
N	
gaussian	
U	
lognorm	
named_pbox	

6.7.3 Classes

UncertainNumber	Uncertain Number class
nInterval	An interval is an uncertain number for which only the endpoints are known, $x = [a, b]$.
Pbox	A probability distribution is a mathematical function that gives the probabilities of occurrence for different possible values of a variable. Probability boxes (p-boxes) represent interval bounds on probability distributions. The left and right quantiles are each stored as a NumPy array containing the percent point function (the inverse of the cumulative distribution function) for *steps* evenly spaced values between 0 and 1. P-boxes can be defined using all the probability distributions that are available through SciPy's statistics library. Naturally, precis probability distributions can be defined by defining a p-box with precise inputs. This means that within probability bounds analysis probability distributions are considered a special case of a p-box with zero width. Distribution-free p-boxes can also be generated when the underlying distribution is unknown but parameters such as the mean, variance or minimum/maximum bounds are known. Such p-boxes make no assumption about the shape of the distribution and instead return bounds expressing all possible distributions that are valid given the known information. Such p-boxes can be constructed making use of Chebyshev, Markov and Cantelli inequalities from probability theory.
Params	

6.7.4 Functions

stochastic_mixture(l_uns[, weights, display])	it could work for either Pbox, distribution, DS structure or Intervals
known_constraints(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution free p-box based upon the information given.
min_max(→ pyuncertainnumber.pba.pbox_base.Pbox)	Returns a box shaped Pbox. This is equivalent to an nInterval expressed as a Pbox.
min_max_mean(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the minimum, maximum and mean of the variable
min_mean(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the minimum and mean of the variable
min_max_mean_std(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the minimum, maximum, mean and standard deviation of the variable
min_max_mean_var(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the minimum, maximum, mean and standard deviation of the variable
min_max_mode(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the minimum, maximum, and mode of the variable
min_max_median(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the minimum, maximum and median of the variable
min_max_median_is_mode(...)	Generates a distribution-free p-box based upon the minimum, maximum and median/mode of the variable when median = mode.
mean_std(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the mean and standard deviation of the variable
mean_var(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon the mean and variance of the variable
pos_mean_std(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a positive distribution-free p-box based upon the mean and standard deviation of the variable
symmetric_mean_std(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a symmetrix distribution-free p-box based upon the mean and standard deviation of the variable
from_percentiles(→ pyuncertainnumber.pba.pbox_base.Pbox)	Generates a distribution-free p-box based upon percentiles of the variable
KS_bounds(...)	construct free pbox from sample data by Kolmogorov-Smirnoff confidence bounds
fit(method, family, data)	top-level fit from data
wc_interval(bound)	wildcard scalar interval
_get_bounds(dist_family, *args[, steps])	from distribution specification to define the lower and upper bounds of the p-box
_bound_pcdf(dist_family, *args[, steps])	bound the parametric CDF
makePbox(func)	
norm(*args)	
lognormal(mean, var[, steps])	Creates a p-box for the lognormal distribution
alpha(*args)	
anglit(*args)	
argus(*args)	

continues on next page

Table 5 – continued from previous page

arcsine(*args)	
beta(*args[, steps])	Beta distribution
betaprime(*args)	
bradford(*args)	
burr(*args)	
burr12(*args)	
cauchy(*args)	
chi(*args)	
chi2(*args)	
cosine(*args)	
crystalball(*args)	
dgamma(*args)	
dweibull(*args)	
erlang(*args)	
expon(*args)	
exponnorm(*args)	
exponweib(*args)	
exponpow(*args)	
f(*args)	
fatiguelife(*args)	
fisk(*args)	
foldcauchy(*args)	
foldnorm(mu, s[, steps])	
genlogistic(*args)	
gennorm(*args)	
genpareto(*args)	

continues on next page

Table 5 – continued from previous page

genexpon(*args)
genextreme(*args)
gausshyper(*args)
gamma(*args)
gengamma(*args)
genhalflogistic(*args)
geninvgauss(*args)
gompertz(*args)
gumbel_r(*args)
gumbel_l(*args)
halfcauchy(*args)
halflogistic(*args)
halfnorm(*args)
halfgennorm(*args)
hypsecant(*args)
invgamma(*args)
invgauss(*args)
invweibull(*args)
irwinhall(*args)
jf_skew_t(*args)
johnsonsb(*args)
johnsonsu(*args)
kappa4(*args)
kappa3(*args)
ksone(*args)
kstwo(*args)

continues on next page

Table 5 – continued from previous page

kstwobign(*args)
laplace(*args)
laplace_asymmetric(*args)
levy(*args)
levy_l(*args)
levy_stable(*args)
logistic(*args)
loggamma(*args)
loglaplace(*args)
loguniform(*args)
lomax(*args)
maxwell(*args)
mielke(*args)
moyal(*args)
nakagami(*args)
ncx2(*args)
ncf(*args)
nct(*args)
norminvgauss(*args)
pareto(*args)
pearson3(*args)
powerlaw(*args)
powerlognorm(*args)
powernorm(*args)
rdist(*args)
rayleigh(*args)

continues on next page

Table 5 – continued from previous page

rel_breitwigner(*args)	
rice(*args)	
recipinvgauss(*args)	
semicircular(*args)	
skewcauchy(*args)	
skewnorm(*args)	
studentized_range(*args)	
t(*args)	
trapezoid(*args)	
triang(*args)	
truncexpon(*args)	
truncnorm(left, right[, mean, stddev, steps])	
truncpareto(*args)	
truncweibull_min(*args)	
tukeylambda(*args)	
uniform_sps(*args)	
vonmises(*args)	
vonmises_line(*args)	
wald(*args)	
weibull_min(*args)	
weibull_max(*args)	
wrapcauchy(*args)	
trapz(a, b, c, d[, steps])	
uniform(a, b[, steps])	special case of Uniform distribution as
weibull(*args[, steps])	
KM(k, m[, steps])	

continues on next page

Table 5 – continued from previous page

KN(k, n[, steps])	
bernoulli(*args)	
betabinom(*args)	
betanbinom(*args)	
binom(*args)	
boltzmann(*args)	
dlaplace(*args)	
geom(*args)	
hypergeom(*args)	
logser(*args)	
nbinom(*args)	
nchypergeom_fisher(*args)	
nchypergeom_wallenius(*args)	
nhypergeom(*args)	
planck(*args)	
poisson(*args)	
randint(*args)	
skellam(*args)	
yulesimon(*args)	
zipf(*args)	
zipfian(*args)	
hedge_interpret(...)	interpret linguistic hedge words into UncertainNumber objects

6.7.5 Package Contents

class pyuncertainnumber.**UncertainNumber**

Uncertain Number class

Parameters

- **bounds;** (-)

- **distribution_parameters** (-) – a list of the distribution family and its parameters; e.g. [‘norm’, [0, 1]];
- **pbox_initialisation** (-) – a list of the distribution family and its parameters; e.g. [‘norm’, ([0,1], [3,4])];
- **naked_value** (-) – the deterministic numeric representation of the UN object, which shall be linked with the ‘pba’ or *Intervals* package

Example

```
>>> UncertainNumber(name="velocity", symbol="v", units="m/s", bounds=[1, 2])
```

name: str = None

symbol: str = None

units: Type[any] = None

_Q: Type[any] = None

uncertainty_type:
Type[*pyuncertainnumber.characterisation.uncertainty_types.Uncertainty_types*] = None

essence: str = None

masses: list[float] = None

bounds: List[float] | str = None

distribution_parameters: list[str, float | int] = None

pbox_parameters: list[str, Sequence[*pyuncertainnumber.pba.interval.Interval*]] = None

hedge: str = None

_construct: Type[any] = None

naked_value: float = None

p_flag: bool = True

measurand: str = None

nature: str = None

provenence: str = None

justification: str = None

structure: str = None

security: str = None

ensemble: Type[*pyuncertainnumber.characterisation.ensemble.Ensemble*] = None

variability: str = None

dependence: str = None

uncertainty: str = None

instances = []

_samples: pyuncertainnumber.characterisation.utils.np.ndarray | list = None

parameterised_pbox_specification()

__post_init__()

the de facto initialisation method for the core math objects of the UN class
caveat:
user needs to by themselves figure out the correct shape of the 'distribution_parameters', such as ['uniform', [1,2]]

static match_pbox(*keyword*, *parameters*)

match the distribution keyword from the initialisation to create the underlying distribution object
Parameters

- **keyword** (-) – (str) the distribution keyword
- **parameters** (-) – (list) the parameters of the distribution

init_check()

check if the UN initialisation specification is correct

> **Note**
>
> a lot of things to double check. keep an growing list: 1. unit 2. hedge: user cannot speficy both 'hedge' and 'bounds'. 'bounds' takes precedence.

__str__()

the verbose user-friendly string representation .. note:

```
this has nothing to do with the logic of JSON serialisation
ergo, do whatever you fancy;
```

__repr__() → str

concise __repr__

describe(*type='verbose'*)

print out a verbose description of the uncertain number

_get_concise_representation()

get a concise representation of the UN object

ci()

get 95% range confidence interval

display(**kwargs*)

quick plot of the uncertain number object

property construct

classmethod from_hedge(*hedged_language*)

create an Uncertain Number from hedged language

> **Note**
>
> # if interval or pbox, to be implemented later on # currently only Interval is supported

classmethod fromConstruct(*construct*)

create an Uncertain Number from a construct object

classmethod fromDistribution(*D*, ***kwargs*)

create an Uncertain Number from specification of distribution

Parameters

- **D** (-) – Distribution object
- **dist_family** (*str*) – the distribution family
- **dist_params** (*list, tuple or string*) – the distribution parameters

classmethod from_Interval(*u*)

classmethod from_pbox(*p*)

genenal from pbox

classmethod from_ds(*ds*)

classmethod from_sps(*sps_dist*)

create an UN object from a parametric scipy.stats dist object #! it seems that a function will suffice :param - sps_dist: scipy.stats dist object

> **Note**
>
> - sps_dist –> UN.Distribution object

sqrt()

__add__(*other*)

add two uncertain numbers

__radd__(*other*)

__sub__(*other*)

__mul__(*other*)

multiply two uncertain numbers

__rmul__(*other*)

__truediv__(*other*)

divide two uncertain numbers

__rtruediv__(*other*)

__pow__(*other*)

power of two uncertain numbers

classmethod _toIntervalBackend(*vars=None*) → pyuncertainnumber.characterisation.utils.np.array

transform any UN object to an *interval* #! currently in use # TODO think if use Marco's Interval Vector object

question:

- what is the *interval* representation: list, nd.array or Interval object?

Returns

- 2D np.array representation for all the interval-typed UNs

classmethod _IntervaltoCompBackend(*vars*)

convert the interval-tupe UNs instantiated to the computational backend

> **Note**
>
> - it will automatically convert all the UN objects in array-like to the computational backend
> - essentially vars shall be all interval-typed UNs by now

Returns

- nd.array or Marco's Interval object

thoughts:

- if Marco's, then we'd use *intervalise* func to get all interval objects

and then to create another func to convert the interval objects to np.array to do endpoints method

JSON_dump(*filename='UN_data.json'*)

the JSON serialisation of the UN object into the filesystem

random(*size=None*)

Generate random samples from the distribution.

ppf(*q=None*)

"Calculate the percent point function (inverse of CDF) at quantile q.

pyuncertainnumber.**stochastic_mixture**(*l_uns*, *weights=None*, *display=False*, ***kwargs*)

it could work for either Pbox, distribution, DS structure or Intervals

Parameters

- **l_un** (-) – list of uncertain numbers
- **weights** (-) – list of weights
- **display** (-) – boolean for plotting

TODO mix types later .. note:: - currently only accepts same type objects

pyuncertainnumber.**known_constraints**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *median:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *mode:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *std:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *var:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *cv:* pyuncertainnumber.pba.interval.Interval | *float* | *int* | *None* = *None*, *percentiles:* *dict[*pyuncertainnumber.pba.interval.Interval | *float* | *int]* | *None* = *None*, *debug: bool = False*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution free p-box based upon the information given. This function works by calculating every possible non-parametric p-box that can be generated using the information provided. The returned p-box is the intersection of these p-boxes.

Parameters:

`minimum`: Minimum value of the variable `maximum`: Maximum value of the variable `mean`: Mean value of the variable `median`: Median value of the variable `mode`: Mode value of the variable `std`: Standard deviation of the variable `var`: Variance of the variable `cv`: Coefficient of variation of the variable `percentiles`: Dictionary of percentiles and their values (e.g. {0.1: 1, 0.5: 2, 0.9: nInterval(3,4)}) `steps`: Number of steps to use in the p-box

Error

`ValueError`: If any of the arguments are not consistent with each other. (i.e. if `std` and `var` are both given, but `std != sqrt(var)`)

Returns:

`Pbox`: Imposition of possible p-boxes

pyuncertainnumber.**min_max**(*a:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *b:* pyuncertainnumber.pba.interval.Interval | *float* | *int* = *None*, *steps=Params.steps*, *shape='box'*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Returns a box shaped Pbox. This is equivalent to an nInterval expressed as a Pbox.

Parameters:

`a` : Left side of box `b`: Right side of box

Returns:

`Pbox`

pyuncertainnumber.**min_max_mean**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum, maximum and mean of the variable

Parameters:

`minimum` : minimum value of the variable

`maximum` : maximum value of the variable

`mean` : mean value of the variable

Returns:

`Pbox`

pyuncertainnumber.**min_mean**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps=Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum and mean of the variable

Parameters:

`minimum` : minimum value of the variable

`mean` : mean value of the variable

Returns:

`Pbox`

pyuncertainnumber.**min_max_mean_std**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *std:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum, maximum, mean and standard deviation of the variable

Parameters

`minimum` : minimum value of the variable `maximum` : maximum value of the variable `mean` : mean value of the variable `std` :standard deviation of the variable

Returns

`Pbox`

See also

min_max_mean_var()

pyuncertainnumber.**min_max_mean_var**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *var:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum, maximum, mean and standard deviation of the variable

Parameters

`minimum` : minimum value of the variable `maximum` : maximum value of the variable `mean` : mean value of the variable `var` :variance of the variable

Returns

`Pbox`

Implementation

Equivalent to `min_max_mean_std(minimum,maximum,mean,np.sqrt(var))`

See also

min_max_mean_std()

pyuncertainnumber.**min_max_mode**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *mode:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum, maximum, and mode of the variable

Parameters:

`minimum` : minimum value of the variable

`maximum` : maximum value of the variable

`mode` : mode value of the variable

Returns:

`Pbox`

pyuncertainnumber.**min_max_median**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *median:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum, maximum and median of the variable

Parameters:

`minimum` : minimum value of the variable

`maximum` : maximum value of the variable

`median` : median value of the variable

Returns:

`Pbox`

pyuncertainnumber.**min_max_median_is_mode**(*minimum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *maximum:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *m:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the minimum, maximum and median/mode of the variable when median = mode.

Parameters:

`minimum` : minimum value of the variable

`maximum` : maximum value of the variable

`m` : m = median = mode value of the variable

Returns:

`Pbox`

pyuncertainnumber.**mean_std**(*mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *std:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps=Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the mean and standard deviation of the variable

Parameters:

`mean` : mean of the variable

`std` : standard deviation of the variable

Returns:

`Pbox`

pyuncertainnumber.**mean_var**(*mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *var:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps=Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon the mean and variance of the variable

Equivalent to *mean_std(mean,np.sqrt(var))*

Parameters:

`mean` : mean of the variable

`var` : variance of the variable

Returns:

`Pbox`

pyuncertainnumber.**pos_mean_std**(*mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *std:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps=Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a positive distribution-free p-box based upon the mean and standard deviation of the variable

Parameters:

`mean` : mean of the variable

`std` : standard deviation of the variable

Returns:

`Pbox`

pyuncertainnumber.**symmetric_mean_std**(*mean:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *std:* pyuncertainnumber.pba.interval.Interval | *float* | *int*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a symmetrix distribution-free p-box based upon the mean and standard deviation of the variable

Parameters:

`mean` : mean value of the variable `std` : standard deviation of the variable

Returns

`Pbox`

pyuncertainnumber.**from_percentiles**(*percentiles: dict*, *steps: int = Params.steps*) → *pyuncertainnumber.pba.pbox_base.Pbox*

Generates a distribution-free p-box based upon percentiles of the variable

Parameters

`percentiles` : dictionary of percentiles and their values (e.g. {0: 0, 0.1: 1, 0.5: 2, 0.9: nInterval(3,4), 1:5})

`steps` : number of steps to use in the p-box

Important

The percentiles dictionary is of the form {percentile: value}. Where value can either be a number or an nInterval. If value is a number, the percentile is assumed to be a point percentile. If value is an nInterval, the percentile is assumed to be an interval percentile.

Warning

If no keys for 0 and 1 are given, `-np.inf` and `np.inf` are used respectively. This will result in a p-box that is not bounded and raise a warning.

If the percentiles are not increasing, the percentiles will be intersected. This may not be desired behaviour.

Error

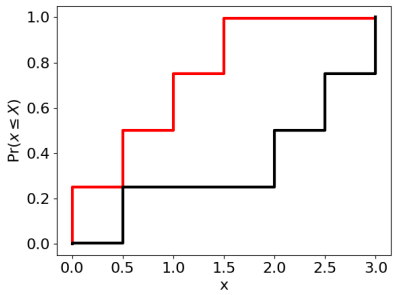
`ValueError`: If any of the percentiles are not between 0 and 1.

Returns

`Pbox`

Example:

```
pba.from_percentiles(
    {0: 0,
    0.25: 0.5,
    0.5: pba.I(1,2),
    0.75: pba.I(1.5,2.5),
    1: 3}
).show()
```



pyuncertainnumber.**KS_bounds**(*s*, *alpha: float*, *display=True*) → Tuple[*pyuncertainnumber.pba.utils.CDF_bundle*, *pyuncertainnumber.pba.utils.CDF_bundle*]

construct free pbox from sample data by Kolmogorov-Smirnoff confidence bounds

Parameters

- **s** (-) – sample data, precise and imprecise
- **dn** (-) – KS critical value at significance level lpha and sample size N;

pyuncertainnumber.**fit**(*method: str*, *family: str*, *data: numpy.ndarray*)

top-level fit from data

Parameters

- **method** (-) – method of fitting, e.g., {'mle' or 'mom'} 'entropy', 'pert', 'fermi', 'bayesian'
- **family** (-) – distribution family to be fitted
- **data** (-) – data to be fitted

Note

- supported family list can be found in xx.

Returns

- the return from the constructors below are *scipy.stats.dist* objects or *UN* objects depending on the decorator

Example

```
>>> pun.fit('mle', 'norm', np.random.normal(0, 1, 100))
```

class pyuncertainnumber.**nInterval**(*left=None*, *right=None*)

An interval is an uncertain number for which only the endpoints are known, $x = [a, b]$. This is interpreted as x being between a and b but with no more information about the value of x.

Intervals embody epistemic uncertainty within PBA.

Creation

Intervals can be created using either of the following:

```
>>> pba.Interval(0,1)
Interval [0,1]
>>> pba.I(2,3)
Interval [2,3]
```

Tip

The shorthand `I` is an alias for `Interval`

Intervals can also be created from a single value $\pm$ half-width:

```
>>> pba.PM(0,1)
Interval [-1,1]
```

By default intervals are displayed as `Interval [a,b]` where `a` and `b` are the left and right endpoints respectively. This can be changed using the **`interval.pm_repr`_** and **`interval.lr_repr`_** functions.

Arithmetic

For two intervals [a,b] and [c,d] the following arithmetic operations are defined:

Addition

$[a, b] + [c, d] = [a + c, b + d]$

Subtraction

$[a, b] - [c, d] = [a - d, b - c]$

Multiplication

$[a, b] * [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$

Division

$[a,b]/[c,d] = [a,b] * \frac{1}{[c,d]} \equiv [\min(a/c, a/d, b/c, b/d), \max(a/c, a/d, b/c, b/d)]$

Alternative arithmetic methods are described in *interval.add*, *interval.sub*, *interval.mul*, *interval.div*.

_left = None

_right = None

property left

property right

__repr__() → str

__str__() → str

__format__(*format_spec: str*) → str

__iter__()

__len__()

__radd__(*left*)

__sub__(*other*)

__rsub__(*other*)

__neg__()

__mul__(*other*)

__rmul__(*other*)

__truediv__(*other*)

__rtruediv__(*other*)

__pow__(*other*)

__rpow__(*left*)

__lt__(*other*)

__rgt__(*other*)

__eq__(*other*)

==

__gt__(*other*)

>

__rlt__(*other*)

__ne__(*other*)

!=

__le__(*other*)

<=

__ge__(*other*)

__bool__()

__abs__()

__contains__(*other*)

add(*other*, *method=None*)

Adds the interval and another object together.

Args:

`other`: The interval or numeric value to be added. This value must be transformable into an Interval object.

Methods:

p - perfect arithmetic $[a, b] + [c, d] = [a + c, b + d]$

o - opposite arithmetic $[a, b] + [c, d] = [a + d, b + c]$

None, i, f - Standard interval arithmetic is used.

Returns:

`Interval`

__add__(*other*)

padd(*other*)

> **Warning**
>
> This method is deprecated. Use add(other, method='p') instead.

oadd(*other*)

> **Warning**
>
> This method is deprecated. Use add(other, method='o') instead.

sub(*other*, *method=None*)

Subtracts other from self.

Args:

`other`: The interval or numeric value to be subracted. This value must be transformable into an Interval object.

Methods:

`p`: perfect arithmetic $a + b = [a.left - b.left, a.right - b.right]$

`o`: opposite arithmetic $a + b = [a.left - b.right, a.right - b.left]$

None, i, f - Standard interval arithmetic is used.

Returns:

`Interval`

psub(*other*)

> **Warning**
>
> Depreciated use self.sub(other, method = 'p') instead

osub(*other*)

> **Warning**
>
> Depreciated use self.sub(other, method = 'o') instead

mul(*other*, *method=None*)

Multiplies self by other.

Args:

`other`: The interval or numeric value to be multiplied. This value must be transformable into an Interval object.

Methods:

`p`: perfect arithmetic $[a,b],[c,d] = [a*c, b*d]$

`o`: opposite arithmetic $[a,b],[c,d] = [a*d, b*c]$

None, i, f - Standard interval arithmetic is used.

Returns:

Interval: The result of the multiplication.

pmul(*other*)

> **Warning**
>
> Depreciated use self.mul(other, method = 'p') instead

omul(*other*)

> **Warning**
>
> Depreciated use self.mul(other, method = 'o') instead

div(*other*, *method=None*)

Divides self by other

If $0 \in other$ it returns a division by zero error

Args:

`other` (Interval or numeric): The interval or numeric value to be multiplied. This value must be transformable into an Interval object.

Methods:

`p`: perfect arithmetic $[a,b],[c,d] = [a*1/c, b*1/d]$

`o`: opposite arithmetic $[a,b],[c,d] = [a*1/d, b*1/c]$

`None`, `i`, `f` - Standard interval arithmetic is used.

> **Implementation**
>
> ```
> >>> self.add(1/other, method = method)
> ```

Error

If $0 \in [a, b]$ it returns a division by zero error

pdiv(*other*)

Warning

Depreciated use self.div(other, method = 'p') instead

odiv(*other*)

Warning

Depreciated use self.div(other, method = 'o') instead

recip()

Calculates the reciprocle of the interval.

Returns:
`Interval`: Equal to $[1/b, 1/a]$

Example:

```
>>> pba.Interval(2,4).recip()
Interval [0.25, 0.5]
```

Error

If $0 \in [a, b]$ it returns a division by zero error

equiv(*other:* Interval) → bool

Checks whether two intervals are equivalent.

Parameters:
`other`: The interval to check against.

Returns `True` **if**:
`self.left == other.right` and `self.right == other.right`

`False` otherwise.

Error

`TypeError`: If `other` is not an instance of `Interval`

See also

`is_same_as()`

Examples:

```
>>> a = Interval(0,1)
>>> b = Interval(0.5,1.5)
>>> c = I(0,1)
>>> a.equiv(b)
False
>>> a.equiv(c)
True
```

property lo

Returns:
self.left

> **Tip**
>
> This function is redundant but exists to match Pbox class for possible internal reasons.

property hi

Returns:
self.right

> **Tip**
>
> This function is redundant but exists to match Pbox class for possible internal reasons.

width() → float

Returns:
float: The width of the interval, $\text{right} - \text{left}$

Example:

```
>>> pba.Interval(0,3).width()
3
```

halfwidth() → float

Returns:
float: The half-width of the interval, $(\text{right} - \text{left})/2$

Example:

```
>>> pba.Interval(0,3).halfwidth()
1.5
```

> **Implementation**
>
> ```
> >>> self.width()/2
> ```

midpoint() → float

Returns the midpoint of the interval

Note

- this serves as the deterministic value representation of the interval, a.k.a. the naked value for an interval

Returns
The midpoint of the interval, $(\text{right} + \text{left})/2$

Return type
`float`

Example:

```
>>> pba.Interval(0,2).midpoint()
1.0
```

`to_logical()`

Turns the interval into a logical interval, this is done by chacking the truth value of the ends of the interval

Returns:
`Logical`: The logical interval

Implementation

```
>>> left = self.left.__bool__()
>>> right = self.right.__bool__()
>>> Logical(left,right)
```

`env`(*other: list* | Interval) → *Interval*

Calculates the envelope between two intervals

Parameters:
`other` : Interval or list. The interval to envelope with self

Hint

If other is a list then the envelope is calculated between self and each element of the list. In this case the envelope is calculated recursively and pba.envelope() may be more efficient.

Important

If other is a Pbox then `Pbox.env()` is called

See also

`pba.core.envelope`_

`pba.pbox.Pbox.env`_

Returns:
`Interval`: The envelope of self and other

straddles(*N: int | float* | Interval, *endpoints: bool = True*) → bool

Parameters:

`N`: Number to check. If N is an interval checks whether the whole interval is within self.

`endpoints`: Whether to include the endpoints within the check

Returns `True` **if**:

$\text{left} \leq N \leq \text{right}$ (Assuming `endpoints=True`).

For interval values. $\text{left} \leq N.left \leq \text{right}$ and $\text{left} \leq N.right \leq \text{right}$ (Assuming `endpoints=True`).

`False` otherwise.

Tip

`N in self` is equivalent to `self.straddles(N)`

straddles_zero(*endpoints=True*)

Checks whether 0 is within the interval

Implementation

Equivalent to `self.straddles(0,endpoints)`

See also

interval.straddles

intersection(*other:* Interval | *list*) → *Interval*

Calculates the intersection between intervals

Parameters:

`other`: The interval to intersect with self. If an interval is not given will try to cast as an interval. If a list is given will calculate the intersection between self and each element of the list.

Returns:

`Interval`: The intersection of self and other. If no intersection is found returns `None`

Example:

```
>>> a = Interval(0,1)
>>> b = Interval(0.5,1.5)
>>> a.intersection(b)
Interval [0.5, 1]
```

exp()

log()

sqrt()

display(*title=''*, *ax=None*, *style='band'*, ***kwargs*)

sample(*seed=None*, *numpy_rng: numpy.random.Generator = None*) → float

Generate a random sample within the interval.

Parameters:

`seed` (int, optional): Seed value for random number generation. Defaults to None.

`numpy_rng` (numpy.random.Generator, optional): Numpy random number generator. Defaults to None.

Returns:

`float`: Random sample within the interval.

Implementation

If `numpy_rng` is given:

```
>>> numpy_rng.uniform(self.left, self.right)
```

Otherwise the following is used:

```
>>> import random
>>> random.seed(seed)
>>> self.left + random.random() * self.width()
```

Examples:

```
>>> pba.Interval(0,1).sample()
0.6160988752201705
>>> pba.I(0,1).sample(seed = 1)
0.13436424411240122
```

If a numpy random number generator is given then it is used instead of the default python random number generator. It has to be initialised first.

```
>>> import numpy as np
>>> rng = np.random.default_rng(seed = 0)
>>> pba.I(0,1).sample(numpy_rng = rng)
0.6369616873214543
```

round()

outward rounding operation for an interval object

classmethod from_midwith(*midpoint: float*, *halfwidth: float*) → *Interval*

Creates an Interval object from a midpoint and half-width.

Parameters

- **midpoint** (-) – The midpoint of the interval.
- **halfwidth** (-) – The half-width of the interval.

Returns

The interval with midpoint and half-width.

Return type

- Interval

Example

```
>>> pba.Interval.from_midwith(0,1)
Interval [-1, 1]
```

class pyuncertainnumber.**Pbox**(*left=None*, *right=None*, *steps=None*, *shape=None*, *mean_left=None*, *mean_right=None*, *var_left=None*, *var_right=None*, *interpolation='linear'*)

A probability distribution is a mathematical function that gives the probabilities of occurrence for different possible values of a variable. Probability boxes (p-boxes) represent interval bounds on probability distributions. The left and right quantiles are each stored as a NumPy array containing the percent point function (the inverse of the cumulative distribution function) for *steps* evenly spaced values between 0 and 1. P-boxes can be defined using all the probability distributions that are available through SciPy's statistics library. Naturally, precis probability distributions can be defined by defining a p-box with precise inputs. This means that within probability bounds analysis probability distributions are considered a special case of a p-box with zero width. Distribution-free p-boxes can also be generated when the underlying distribution is unknown but parameters such as the mean, variance or minimum/maximum bounds are known. Such p-boxes make no assumption about the shape of the distribution and instead return bounds expressing all possible distributions that are valid given the known information. Such p-boxes can be constructed making use of Chebyshev, Markov and Cantelli inequalities from probability theory.

shape = None

__repr__()

__str__

__iter__()

__neg__()

__lt__(*other*)

__rlt__(*other*)

__le__(*other*)

__rle__(*other*)

__gt__(*other*)

__rgt__(*other*)

__ge__(*other*)

__rge__(*other*)

__and__(*other*)

__rand__(*other*)

__or__(*other*)

__ror__(*other*)

__add__(*other*)

__radd__(*other*)

__sub__(*other*)

__rsub__(*other*)

__mul__(*other*)

__rmul__(*other*)

__pow__(*other*)

__rpow__(*other*)

__truediv__(*other*)

__rtruediv__(*other*)

property range

leslie defined range property

property lo

Returns the left-most value in the interval

property hi

Returns the right-most value in the interval

get_range()

get the quantile range of either a pbox or a distribution

_computemoments()

_checkmoments()

cutv(*x*)

get the bounds on the cumulative probability associated with any x-value

cuth(*p=0.5*)

get the bounds on the quantile at any particular probability level

outer_approximate(*n=100*)

outer approximation of a p-box

> **Note**
>
> - *the_interval_list* will have length one less than that of *p_values* (i.e. 100 and 99)

_unary(**args*, *function=lambda x: ...*)

for monotonic unary functions only

exp()

sqrt()

recip()

static check_dependency(*method*)

constant_shape_check()

a helper drop in for define binary ops

steps_check(*other*)

add(*other: Self | pyuncertainnumber.pba.interval.Interval | float | int*, *method='f'*) → Self

addtion of uncertain numbers with the defined dependency method

pow(*other: Self | pyuncertainnumber.pba.interval.Interval | float | int*, *method='f'*) → Self

Raises a p-box to the power of other using the defined dependency method

sub(*other*, *method='f'*)

mul(*other*, *method='f'*)

Multiplication of uncertain numbers with the defined dependency method

div(*other*, *method='f'*)

lt(*other*, *method='f'*)

le(*other*, *method='f'*)

gt(*other*, *method='f'*)

ge(*other*, *method='f'*)

min(*other*, *method='f'*)

Returns a new Pbox object that represents the element-wise minimum of two Pboxes.

Parameters

- **other** (-) – Another Pbox object or a numeric value.
- **method** (-) – Calculation method to determine the minimum. Can be one of 'f', 'p', 'o', 'i'.

Returns

Pbox

max(*other*, *method='f'*)

truncate(*a*, *b*, *method='f'*)

Equivalent to self.min(a,method).max(b,method)

env(*other*)

Computes the envelope of two Pboxes.

Parameters: - other: Pbox or numeric value

The other Pbox or numeric value to compute the envelope with.

Returns: - Pbox

The envelope Pbox.

Raises: - ArithmeticError: If both Pboxes have different number of steps.

imp(*other*)

Returns the imposition of self with other pbox

Note

- binary imposition between two pboxes only

logicaland(*other*, *method='f'*)

logicalor(*other*, *method='f'*)

get_interval(**args*) → *pyuncertainnumber.pba.interval.Interval*

get_probability(*val*) → *pyuncertainnumber.pba.interval.Interval*

summary() → str

mean() → *pyuncertainnumber.pba.interval.Interval*

Returns the mean of the pbox

median() → *pyuncertainnumber.pba.interval.Interval*

Returns the median of the distribution

support() → *pyuncertainnumber.pba.interval.Interval*

get_x()

returns the x values for plotting

get_y()

returns the y values for plotting

straddles(*N*, *endpoints=True*) → bool

Parameters

- **N** (*numeric*) – Number to check
- **endpoints** (*bool*) – Whether to include the endpoints within the check

Returns

- *True* – If left $\leq N \leq$ right (Assuming *endpoints=True*)
- *False* – Otherwise

straddles_zero(*endpoints=True*) → bool

Checks whether 0 is within the p-box

show(*figax=None*, *now=True*, *title=''*, *x_axis_label='x'*, ***kwargs*)

legacy plotting function

display(*title=''*, *ax=None*, *style='band'*, *fill_color='lightgray'*, *bound_colors=None*, ***kwargs*)

default plotting function

to_ds_old(*discretisation=Params.steps*)

convert to ds object

Note

- without outer approximation

to_ds(*discretisation=Params.steps*)

convert to ds object

class pyuncertainnumber.**Params**

hedge_cofficients

steps = 200

many = 2000

p_values

p_lboundary = 0.0001

p_hboundary = 0.9999

scott_hedged_interpretation

user_hedged_interpretation

result_path = './results/'

hw = 0.5

pyuncertainnumber.**wc_interval**(*bound*)

wildcard scalar interval

pyuncertainnumber.**_get_bounds**(*dist_family*, **args*, *steps=Params.steps*)

from distribution specification to define the lower and upper bounds of the p-box

Parameters

dist_family (-) – (str) the name of the distribution

pyuncertainnumber.**_bound_pcdf**(*dist_family*, **args*, *steps=Params.steps*)

bound the parametric CDF

> **Note**
>
> - only support fully bounded parameters

pyuncertainnumber.**makePbox**(*func*)

pyuncertainnumber.**norm**(**args*)

pyuncertainnumber.**lognormal**(*mean*, *var*, *steps=Params.steps*)

Creates a p-box for the lognormal distribution

Note: the parameters used are the mean and variance of the lognormal distribution not the mean and variance of the underlying normal See: *[1]<https://en.wikipedia.org/wiki/Log-normal_distribution#Generation_and_parameters>*
[2]<https://stackoverflow.com/questions/51906063/distribution-mean-and-standard-deviation-using-scipy-stats>

Parameters

- **mean** – mean of the lognormal distribution
- **var** – variance of the lognormal distribution

Return type

Pbox

pyuncertainnumber.**alpha**(**args*)

pyuncertainnumber.**anglit**(**args*)

pyuncertainnumber.**argus**(**args*)

pyuncertainnumber.**arcsine**(**args*)

pyuncertainnumber.**beta**(**args*, *steps=Params.steps*)

Beta distribution

pyuncertainnumber.**betaprime**(**args*)

pyuncertainnumber.**bradford**(**args*)

pyuncertainnumber.**burr**(**args*)

pyuncertainnumber.**burr12**(**args*)

pyuncertainnumber.**cauchy**(**args*)

pyuncertainnumber.**chi**(**args*)

pyuncertainnumber.**chi2**(**args*)

pyuncertainnumber.**cosine**(**args*)

pyuncertainnumber.**crystalball**(**args*)

pyuncertainnumber.**dgamma**(**args*)

pyuncertainnumber.**dweibull**(**args*)

pyuncertainnumber.**erlang**(**args*)

pyuncertainnumber.**expon**(**args*)

pyuncertainnumber.**exponnorm**(**args*)

pyuncertainnumber.**exponweib**(**args*)

pyuncertainnumber.**exponpow**(**args*)

pyuncertainnumber.**f**(**args*)

pyuncertainnumber.**fatiguelife**(**args*)

pyuncertainnumber.**fisk**(**args*)

pyuncertainnumber.**foldcauchy**(**args*)

pyuncertainnumber.**foldnorm**(*mu*, *s*, *steps=Params.steps*)

pyuncertainnumber.**genlogistic**(**args*)

pyuncertainnumber.**gennorm**(**args*)

pyuncertainnumber.**genpareto**(**args*)

pyuncertainnumber.**genexpon**(**args*)

pyuncertainnumber.**genextreme**(**args*)

pyuncertainnumber.**gausshyper**(**args*)

pyuncertainnumber.**gamma**(**args*)

pyuncertainnumber.**gengamma**(**args*)

pyuncertainnumber.**genhalflogistic**(**args*)

pyuncertainnumber.**geninvgauss**(**args*)

pyuncertainnumber.**gompertz**(**args*)

pyuncertainnumber.**gumbel_r**(**args*)

pyuncertainnumber.**gumbel_l**(**args*)

pyuncertainnumber.**halfcauchy**(**args*)

pyuncertainnumber.**halflogistic**(**args*)

pyuncertainnumber.**halfnorm**(**args*)

pyuncertainnumber.**halfgennorm**(**args*)

pyuncertainnumber.**hypsecant**(**args*)

pyuncertainnumber.**invgamma**(**args*)

pyuncertainnumber.**invgauss**(**args*)

pyuncertainnumber.**invweibull**(**args*)

pyuncertainnumber.**irwinhall**(**args*)

pyuncertainnumber.**jf_skew_t**(**args*)

pyuncertainnumber.**johnsonsb**(**args*)

pyuncertainnumber.**johnsonsu**(**args*)

pyuncertainnumber.**kappa4**(**args*)

pyuncertainnumber.**kappa3**(**args*)

pyuncertainnumber.**ksone**(**args*)

pyuncertainnumber.**kstwo**(**args*)

pyuncertainnumber.**kstwobign**(**args*)

pyuncertainnumber.**laplace**(**args*)

pyuncertainnumber.**laplace_asymmetric**(**args*)

pyuncertainnumber.**levy**(**args*)

pyuncertainnumber.**levy_l**(**args*)

pyuncertainnumber.**levy_stable**(**args*)

pyuncertainnumber.**logistic**(**args*)

pyuncertainnumber.**loggamma**(**args*)

pyuncertainnumber.**loglaplace**(**args*)

pyuncertainnumber.**loguniform**(**args*)

pyuncertainnumber.**lomax**(**args*)

pyuncertainnumber.**maxwell**(**args*)

pyuncertainnumber.**mielke**(**args*)

pyuncertainnumber.**moyal**(**args*)

pyuncertainnumber.**nakagami**(**args*)

pyuncertainnumber.**ncx2**(**args*)

pyuncertainnumber.**ncf**(**args*)

pyuncertainnumber.**nct**(**args*)

pyuncertainnumber.**norminvgauss**(**args*)

pyuncertainnumber.**pareto**(**args*)

pyuncertainnumber.**pearson3**(**args*)

pyuncertainnumber.**powerlaw**(**args*)

pyuncertainnumber.**powerlognorm**(**args*)

pyuncertainnumber.**powernorm**(**args*)

pyuncertainnumber.**rdist**(**args*)

pyuncertainnumber.**rayleigh**(**args*)

pyuncertainnumber.**rel_breitwigner**(**args*)

pyuncertainnumber.**rice**(**args*)

pyuncertainnumber.**recipinvgauss**(**args*)

pyuncertainnumber.**semicircular**(**args*)

pyuncertainnumber.**skewcauchy**(**args*)

pyuncertainnumber.**skewnorm**(**args*)

pyuncertainnumber.**studentized_range**(**args*)

pyuncertainnumber.**t**(**args*)

pyuncertainnumber.**trapezoid**(**args*)

pyuncertainnumber.**triang**(**args*)

pyuncertainnumber.**truncexpon**(**args*)

pyuncertainnumber.**truncnorm**(*left*, *right*, *mean=None*, *stddev=None*, *steps=Params.steps*)

pyuncertainnumber.**truncpareto**(**args*)

pyuncertainnumber.**truncweibull_min**(**args*)

pyuncertainnumber.**tukeylambda**(**args*)

pyuncertainnumber.**uniform_sps**(**args*)

pyuncertainnumber.**vonmises**(**args*)

pyuncertainnumber.**vonmises_line**(**args*)

pyuncertainnumber.**wald**(**args*)

pyuncertainnumber.**weibull_min**(**args*)

pyuncertainnumber.**weibull_max**(**args*)

pyuncertainnumber.**wrapcauchy**(**args*)

pyuncertainnumber.**trapz**(*a*, *b*, *c*, *d*, *steps=Params.steps*)

pyuncertainnumber.**uniform**(*a*, *b*, *steps=Params.steps*)

special case of Uniform distribution as Scipy has an unbelivably strange parameterisation than common sense

Parameters

- **a** (-) – (float) lower endpoint
- **b** (-) – (float) upper endpoints

pyuncertainnumber.**weibull**(**args*, *steps=Params.steps*)

pyuncertainnumber.**KM**(*k*, *m*, *steps=Params.steps*)

pyuncertainnumber.**KN**(*k*, *n*, *steps=Params.steps*)

pyuncertainnumber.**bernoulli**(**args*)

pyuncertainnumber.**betabinom**(**args*)

pyuncertainnumber.**betanbinom**(**args*)

pyuncertainnumber.**binom**(**args*)

pyuncertainnumber.**boltzmann**(**args*)

pyuncertainnumber.**dlaplace**(**args*)

pyuncertainnumber.**geom**(**args*)

pyuncertainnumber.**hypergeom**(**args*)

pyuncertainnumber.**logser**(**args*)

pyuncertainnumber.**nbinom**(**args*)

pyuncertainnumber.**nchypergeom_fisher**(**args*)

pyuncertainnumber.**nchypergeom_wallenius**(**args*)

pyuncertainnumber.**nhypergeom**(**args*)

pyuncertainnumber.**planck**(**args*)

pyuncertainnumber.**poisson**(**args*)

pyuncertainnumber.**randint**(**args*)

pyuncertainnumber.**skellam**(**args*)

pyuncertainnumber.**yulesimon**(**args*)

pyuncertainnumber.**zipf**(**args*)

pyuncertainnumber.**zipfian**(**args*)

pyuncertainnumber.**normal**

pyuncertainnumber.**N**

pyuncertainnumber.**gaussian**

pyuncertainnumber.**U**

pyuncertainnumber.**lognorm**

pyuncertainnumber.**named_pbox**

pyuncertainnumber.**hedge_interpret**(*hedge: str*, *return_type='interval'*) → *pyuncertainnumber.pba.interval.Interval* | *pyuncertainnumber.pba.pbox_base.Pbox*

interpret linguistic hedge words into UncertainNumber objects

Parameters

- **hedge** (*str*) – the hedge numerical expression to be interpreted
- **return_type** (*str*) – the type of object to be returned, either 'interval' or 'pbox'

Note

- the return can either be an interval or a pbox object

Example

```
>>> hedge_interpret("about 200", return_type="pbox")
```

6.8 cartesian_product

6.8.1 Functions

cartesian(*arrays)	Computes the Cartesian product of multiple input arrays

6.8.2 Module Contents

cartesian_product.**cartesian**(**arrays*)

Computes the Cartesian product of multiple input arrays

Parameters

***arrays** (-) – Variable number of np.arrays representing the sets of values for each dimension.

signature:

- cartesian(*x:np.array) -> np.ndarray

> **Note**
>
> - The data type of the output array is determined based on the input arrays to ensure compatibility.

Returns

A NumPy array where each row represents one combination from the Cartesian product.

The number of columns equals the number of input arrays.

Return type

- darray

Example

```
>>> x = np.array([1, 2], [3, 4], [5, 6])
>>> y = cartesian(*x)
>>> # Output:
>>> # array([[1, 3, 5],
>>> #         [1, 3, 6],
>>> #         [1, 4, 5],
>>> #         [1, 4, 6],
>>> #         [2, 3, 5],
>>> #         [2, 3, 6],
>>> #         [2, 4, 5],
>>> #         [2, 4, 6]])
```

6.9 sampling_aleatory

6.9.1 Functions

sampling_aleatory_method(...)	Performs uncertainty propagation using Monte Carlo or Latin Hypercube sampling, similar to the *sampling_method*.

6.9.2 Module Contents

sampling_aleatory.**sampling_aleatory_method**(*x: list*, *f: Callable*, *results:* pyuncertainnumber.propagation.utils.Propagation_results = *None*, *n_sam: int = 500*, *method='monte_carlo'*, *save_raw_data='no'*) → *pyuncertainnumber.propagation.utils.Propagation_results*

Performs uncertainty propagation using Monte Carlo or Latin Hypercube sampling, similar to the *sampling_method*.

Parameters

- **x** (-) – A list of *UncertainNumber* objects, each representing an input variable with its associated uncertainty.
- **f** (-) – A callable function that takes a 1D NumPy array of input values and returns the corresponding output(s). Can be None, in which case only samples are generated.
- **results** (-) – An object to store propagation results. Defaults to None, in which case a new *Propagation_results* object is created.
- **n_sam** (-) – The number of samples to generate for the chosen sampling method. Defaults to 500.
- **method** (-) – The sampling method to use. Choose from: - 'monte_carlo': Monte Carlo sampling (random sampling

  from the distributions specified in the UncertainNumber objects)

  - **'latin_hypercube': Latin Hypercube sampling (stratified** sampling for better space coverage)

  Defaults to 'monte_carlo'.
- **save_raw_data** (-) – Whether to save raw data. Options: 'yes', 'no'. Defaults to 'no'.

signature:

sampling_aleatory_method(x: list, f: Callable, results: Propagation_results = None, ...) -> Propagation_results

Note

- If the *f* function returns multiple outputs, the code can accomodate.

Returns

A *Propagation_results* object containing:

- **'raw_data': A dictionary containing raw data (if *save_raw_data* is 'yes'):**
  - 'x': All generated input samples.
  - **'f': Corresponding output values for** each input sample (if *f* is provided).

Return type

- Propagation_results

Raises

- **ValueError** – If an invalid sampling method or *save_raw_data* option is provided.

Example

```
>>> results = sampling_aleatory_method(x=x, f=Fun, n_sam = 300, method = 'monte_
↪carlo', save_raw_data = "no")
```

6.10 extreme_point_func

6.10.1 Functions

extreme_pointX(ranges, signX)	Calculates the extreme points of a set of ranges based on signs.

6.10.2 Module Contents

extreme_point_func.**extreme_pointX**(*ranges*, *signX*)

Calculates the extreme points of a set of ranges based on signs.

Parameters

- **ranges** – A NumPy array of shape (d, 2) representing the ranges (each row is a variable, each column is a bound).
- **signX** – A NumPy array of shape (1, d) representing the signs.

Returns

A NumPy array of shape (2, d) representing the extreme points.

6.11 local_optimisation

6.11.1 Functions

local_optimisation_method(...)	Performs local optimization to find both the minimum and maximum values of a given function, within specified bounds.

6.11.2 Module Contents

local_optimisation.**local_optimisation_method**(*x: numpy.ndarray*, *f: Callable*, *x0: numpy.ndarray = None*, *results:* pyuncertainnumber.propagation.utils.Propagation_results *= None*, *tol_loc: numpy.ndarray = None*, *options_loc: dict = None*, ***, *method_loc='Nelder-Mead'*) → *pyuncertainnumber.propagation.utils.Propagation_results*

Performs local optimization to find both the minimum and maximum values of a given function, within specified bounds. This function utilizes the *scipy.optimize.minimize* function to perform local optimization. Refer to *scipy.optimize.minimize* documentation for available options

args:

x (np.ndarray): A 2D NumPy array where each row represents an input variable and the two columns define its lower and upper bounds (interval).

f (Callable): The objective function to be optimized. It should take a 1D NumPy array
as input and return a scalar value.

x0 (np.ndarray, optional): A 1D or 2D NumPy array representing the initial guess for the
optimization. - If x0 has shape (n,), the same initial values are used for both

minimization and maximization.

- If x0 has shape (2, n), x0[0, :] is used for minimization and x0[1, :] for maximization.

If not provided, the midpoint of each variable's interval is used. Defaults to None.

tol_loc (np.ndarray, optional): Tolerance for termination.

- If tol_loc is a scalar, the same tolerance is used for both minimization and maximization.
- If tol_loc is an array of shape (2,), tol_loc[0] is used for minimization and tol_loc[1] for maximization.

Defaults to None.

options_loc (dict, optional): A dictionary of solver options.

- If options_loc is a dictionary, the same options are used for both minimization and maximization.
- If options_loc is a list of two dictionaries, options_loc[0] is used for minimization and options_loc[1] for maximization.

Refer to *scipy.optimize.minimize* documentation for available options. Defaults to None.

method_loc (str, optional): The optimization method to use (e.g., 'Nelder-Mead', 'COBYLA').
Defaults to 'Nelder-Mead'.

signature:

local_optimisation_method(x:np.ndarray, f:Callable, results:dict = None,
*, x0:np.ndarray = None, tol_loc:np.ndarray = None, options_loc: dict = None, method_loc = 'Nelder-Mead') -> dict

returns:

dict: A dictionary containing the optimization results:

- 'bounds': An np.ndarray of the bounds for the output parameter (if f is not None).
- 'min': Results for minimization, including 'x', 'f', 'message', 'nit', 'nfev', 'final_simplex'.
- 'max': Results for maximization, including 'x', 'f', 'message', 'nit', 'nfev', 'final_simplex'.

example:

```
>>> f = lambda x: x[0] + x[1] + x[2]  # Example function
>>> x_bounds = np.array([[1, 2], [3, 4], [5, 6]])
>>> # Initial guess (same for min and max)
>>> x0 = np.array([1.5, 3.5, 5.5])
>>> # Different tolerances for min and max
>>> tol_loc = np.array([1e-4, 1e-6])
```

(continues on next page)

(continued from previous page)

```
>>> # Different options for min and max
>>> options_loc = [
>>>     {'maxiter': 100},  # Options for minimization
>>>     {'maxiter': 1000}  # Options for maximization
>>>     ]
>>> # Perform optimization
>>> y = local_optimisation_method(x_bounds, f, x0=x0, tol_loc=tol_loc,
>>>                                    options_loc=options_loc)
>>> # Print the results
>>> y.print()
```

6.12 figures_distr_pbox

6.12.1 Attributes

means	
stds	
n_disc	
x	
x0	
y	
y	

6.12.2 Functions

outward_direction(x[, n_disc, tOp, bOt])	
plotPbox_pbox(xL, xR[, p])	Plots a p-box (probability box) using matplotlib.
plotPbox(xL, xR[, p])	Plots a p-box (probability box) using matplotlib.
plot_interval(lower_bound, upper_bound[, color, label])	Plots an interval on a Matplotlib plot.

6.12.3 Module Contents

figures_distr_pbox.**outward_direction**(*x: list*, *n_disc: int | numpy.ndarray = 10*, *tOp: float | numpy.ndarray = 0.999*, *bOt: float | numpy.ndarray = 0.001*)

Parameters

- **x** (*list*) – A list of UncertainNumber objects.
- **f** (*Callable*) – The function to evaluate.
- **results** (*dict*) – A dictionary to store the results (optional).

- **method** (*str*) – The method which will estimate bounds of each combination of focal elements (default is the endpoint)
- **lim_Q** (*np.array*) – Quantile limits for discretization.
- **n_disc** (*int*) – Number of discretization points.

signature:
second_order_propagation_method(x: list, f: Callable, results: dict, method: str, lim_Q: np.array, n_disc: int) -> dict

Notes

Performs second-order uncertainty propagation for mixed uncertain numbers

Returns
A dictionary containing the results

Return type
dict

figures_distr_pbox.**plotPbox_pbox**(*xL*, *xR*, *p=None*)

Plots a p-box (probability box) using matplotlib.

Parameters

- **xL** (*np.ndarray*) – A 1D NumPy array of lower bounds.
- **xR** (*np.ndarray*) – A 1D NumPy array of upper bounds.
- **p** (*np.ndarray, optional*) – A 1D NumPy array of probabilities corresponding to the intervals. Defaults to None, which generates equally spaced probabilities.
- **color** (*str, optional*) – The color of the plot. Defaults to 'k' (black).

figures_distr_pbox.**plotPbox**(*xL*, *xR*, *p=None*)

Plots a p-box (probability box) using matplotlib.

Parameters

- **xL** (*np.ndarray*) – A 1D NumPy array of lower bounds.
- **xR** (*np.ndarray*) – A 1D NumPy array of upper bounds.
- **p** (*np.ndarray, optional*) – A 1D NumPy array of probabilities corresponding to the intervals. Defaults to None, which generates equally spaced probabilities.
- **color** (*str, optional*) – The color of the plot. Defaults to 'k' (black).

figures_distr_pbox.**means**

figures_distr_pbox.**stds**

figures_distr_pbox.**n_disc = 10**

figures_distr_pbox.**x**

figures_distr_pbox.**x0**

figures_distr_pbox.**y**

figures_distr_pbox.**plot_interval**(*lower_bound*, *upper_bound*, *color='blue'*, *label=None*)

Plots an interval on a Matplotlib plot.

Parameters

- **lower_bound** (*float*) – The lower bound of the interval.

- **upper_bound** (*float*) – The upper bound of the interval.
- **color** (*str, optional*) – The color of the interval line. Defaults to 'blue'.
- **label** (*str, optional*) – The label for the interval in the legend. Defaults to None.

figures_distr_pbox.**y**

6.13 genetic_optimisation

6.13.1 Functions

a(x)	
genetic_optimisation_method(...)	Performs both minimisation and maximisation using a genetic algorithm.

6.13.2 Module Contents

genetic_optimisation.**a**(*x*)

genetic_optimisation.**genetic_optimisation_method**(*x_bounds: numpy.ndarray*, *f: Callable*, *results:* pyuncertainnumber.propagation.utils.Propagation_results = *None*, *pop_size=1000*, *n_gen=100*, *tol=0.001*, *n_gen_last=10*, *algorithm_type='NSGA2'*) → *pyuncertainnumber.propagation.utils.Propagation_results*

Performs both minimisation and maximisation using a genetic algorithm.

Parameters

- **x_bounds** – Bounds for decision variables (NumPy array).
- **f** – Objective function to optimize.
- **pop_size** – Population size (int or array of shape (2,)).
- **n_gen** – Maximum number of generations (int or array of shape (2,)).
- **tol** – Tolerance for convergence check (float or array of shape (2,)).
- **n_gen_last** – Number of last generations to consider for convergence (int or array of shape (2,)).
- **algorithm_type** – 'NSGA2' or 'GA' to select the optimisation algorithm (str or array of shape (2,)).

signature:

genetic_optimisation_method(x_bounds: np.ndarray, f: Callable, results:dict,
pop_size=1000, n_gen=100, tol=1e-3, n_gen_last=10, algorithm_type="NSGA2") -> dict

Returns

A dictionary containing the optimisation results:

- 'bounds': An np.ndarray of the bounds for the output parameter (if f is not None).
- 'min': A dictionary with keys 'x', 'f', 'n_gen', and 'n_iter' for minimisation results.

- 'max': A dictionary with keys 'x', 'f', 'n_gen', and 'n_iter' for maximisation results.

Return type
dict

Example

```
>>> # Example usage with different parameters for minimisation and maximisation
>>> f = lambda x: x[0] + x[1] + x[2] # Example function
>>> x_bounds = np.array([[1, 2], [3, 4], [5, 6]])
>>> # Different population sizes for min and max
>>> pop_size = np.array([500, 1500])
>>> # Different number of generations
>>> n_gen = np.array([50, 150])
>>> # Different tolerances
>>> tol = np.array([1e-2, 1e-4])
>>> # Different algorithms
>>> algorithm_type = np.array(["GA", "NSGA2"])
>>> y = genetic_optimisation_method(x_bounds, f, pop_size=pop_size, n_gen=n_gen,
>>>                                 tol=tol, n_gen_last=10, algorithm_
↪type=algorithm_type)
>>> # Print the results
>>> y.print()
```

6.14 first_order_propagation

6.14.1 Functions

imp(X)	Imposition of intervals.
first_order_propagation_method(...)	

6.14.2 Module Contents

first_order_propagation.**imp**(*X*)

Imposition of intervals.

first_order_propagation.**first_order_propagation_method**(*x: list*, *f: Callable = None*, *results:* pyuncertainnumber.propagation.utils.Propagation_results = *None*, *n_disc: int | numpy.ndarray = 10*, *condensation: float | numpy.ndarray = None*, *tOp: float | numpy.ndarray = 0.999*, *bOt: float | numpy.ndarray = 0.001*, *save_raw_data='no'*) → *pyuncertainnumber.propagation.utils.Propagation_results*

Parameters

- **x** (-) – A list of *UncertainNumber* objects representing the uncertain inputs.
- **f** (-) – The function to evaluate.

- **results** (-) – An object to store propagation results. Defaults to None, in which case a new *Propagation_results* object is created.
- **n_disc** (-) – The number of discretization points for each uncertain input. If an integer is provided, it is used for all inputs. If a NumPy array is provided, each element specifies the number of discretization points for the corresponding input. Defaults to 10.
- **condensation** (-) – A parameter or array of parameters to control the condensation of the output p-boxes. Defaults to None.
- **tOp** (-) – Upper threshold or array of thresholds for discretization. Defaults to 0.999.
- **bOt** (-) – Lower threshold or array of thresholds for discretization. Defaults to 0.001.
- **save_raw_data** (-) – Whether to save raw data ('yes' or 'no'). Defaults to 'no'.

signature:

first_order_propagation_method(x: list, f: Callable, results: Propagation_results = None, ...) -> Propagation_results

Notes

- Performs first-order uncertainty propagation for mixed uncertain numbers.
- The function handles different types of uncertain numbers (distributions and p-boxes for this version) and discretizes them with the same number of n_disc.
- It uses the *extremepoints* to determine the signs of the partial derivatives of the function.
- The output p-boxes are constructed by imposing the results from individual input discretizations.
- The *condensation* parameter can be used to reduce the number of intervals in the output p-boxes.

Returns

A *Propagation_results* object containing the results of the
uncertainty propagation. The results include p-boxes representing the output uncertainty.

Return type

Propagation_results

Example

from pyuncertainnumber import UncertainNumber

def Fun(x):

input1= x[0] input2=x[1] input3=x[2] input4=x[3] input5=x[4]

output1 = input1 + input2 + input3 + input4 + input5 output2 = input1 * input2 * input3 * input4 * input5

return np.array([output1, output2])

means = np.array([1, 2, 3, 4, 5]) stds = np.array([0.1, 0.2, 0.3, 0.4, 0.5])

x = [

UncertainNumber(essence = 'distribution', distribution_parameters= ["gaussian",[means[0], stds[0]]]),

from pyuncertainnumber import UncertainNumber

def Fun(x):

input1= x[0] input2=x[1] input3=x[2] input4=x[3] input5=x[4]

output1 = input1 + input2 + input3 + input4 + input5 output2 = input1 * input2 * input3 * input4 * input5

return np.array([output1, output2])

means = np.array([1, 2, 3, 4, 5]) stds = np.array([0.1, 0.2, 0.3, 0.4, 0.5])

x = [
UncertainNumber(essence = 'distribution', distribution_parameters= ["gaussian",[means[0], stds[0]]]),
UncertainNumber(essence = 'distribution', distribution_parameters= ["gaussian",[means[1], stds[1]]]),
UncertainNumber(essence = 'distribution', distribution_parameters= ["gaussian",[means[2], stds[2]]]),
UncertainNumber(essence = 'distribution', distribution_parameters= ["gaussian",[means[3], stds[3]]]),
UncertainNumber(essence = 'distribution', distribution_parameters= ["gaussian",[means[4], stds[4]]]),]
results = first_order_propagation_method(x=x, f=Fun, n_disc= 5)

6.15 second_order_propagation

6.15.1 Functions

second_order_propagation_method(...)

6.15.2 Module Contents

second_order_propagation.**second_order_propagation_method**(*x: list*, *f: Callable = None*, *results:* pyuncertainnumber.propagation.utils.Propagation_results *= None*, *method='endpoints'*, *n_disc: int | numpy.ndarray = 10*, *condensation: float | numpy.ndarray = None*, *tOp: float | numpy.ndarray = 0.999*, *bOt: float | numpy.ndarray = 0.001*, *save_raw_data='no'*) → *pyuncertainnumber.propagation.utils.Propagation_results*

Parameters

- **x** (-) – A list of *UncertainNumber* objects representing the uncertain inputs.
- **f** (-) – The function to evaluate.
- **results** (-) – An object to store propagation results. Defaults to None, in which case a new *Propagation_results* object is created.
- **method** (-) – The method used to estimate the bounds of each combination of focal elements. Can be either 'endpoints' or 'extremepoints'. Defaults to 'endpoints'.
- **n_disc** (-) – The number of discretization points for each uncertain input. If an integer is provided, it is used for all inputs. If a NumPy array is provided, each element specifies the number of discretization points for the corresponding input. Defaults to 10.
- **condensation** (-) – A parameter or array of parameters to control the condensation of the output p-boxes. Defaults to None.
- **tOp** (-) – Upper threshold or array of thresholds for discretization. Defaults to 0.999.
- **bOt** (-) – Lower threshold or array of thresholds for discretization. Defaults to 0.001.
- **save_raw_data** (-) – Whether to save raw data ('yes' or 'no'). Defaults to 'no'.

signature:
second_order_propagation_method(x: list, f: Callable, results: Propagation_results = None, ...) -> Propagation_results

Notes

- Performs second-order uncertainty propagation for mixed uncertain numbers.
- The function handles different types of uncertain numbers (distributions, intervals, p-boxes) and discretizes them accordingly.
- It generates combinations of focal elements from the discretized uncertain inputs.
- For the 'endpoints' method, it evaluates the function at all combinations of endpoints of the focal elements.
- For the 'extremepoints' method, it uses the *extremepoints_method* to determine the signs of the partial derivatives and evaluates the function at the extreme points.
- The output p-boxes are constructed by considering the minimum and maximum values obtained from the function evaluations.
- The *condensation* parameter can be used to reduce the number of intervals in the output p-boxes.

Returns

A *Propagation_results* object containing:

- **raw_data (dict): Dictionary containing raw data shared across output(s):**
  - x (np.ndarray): Input values.
  - f (np.ndarray): Output values.
  - **min (np.ndarray): Array of dictionaries, one for each output,** containing 'f' for the minimum of that output.
  - **max (np.ndarray): Array of dictionaries, one for each output,** containing 'f' for the maximum of that output.
  - bounds (np.ndarray): 2D array of lower and upper bounds for each output.

Return type

Propagation_results

Example

from pyuncertainnumber import UncertainNumber

def Fun(x):

input1= x[0] input2=x[1] input3=x[2] input4=x[3] input5=x[4]

output1 = input1 + input2 + input3 + input4 + input5 output2 = input1 * input2 * input3 * input4 * input5

return np.array([output1, output2])

means = np.array([1, 2, 3, 4, 5]) stds = np.array([0.1, 0.2, 0.3, 0.4, 0.5])

x = [

UncertainNumber(essence = 'distribution', distribution_parameters= ["gaussian",[means[0], stds[0]]]),

UncertainNumber(essence = 'interval', bounds= [means[1]-2* stds[1], means[1]+2* stds[1]]), UncertainNumber(essence = 'interval', bounds= [means[2]-2* stds[2], means[2]+2* stds[2]]), UncertainNumber(essence = 'interval', bounds= [means[3]-2* stds[3], means[3]+2* stds[3]]), UncertainNumber(essence = 'interval', bounds= [means[4]-2* stds[4], means[4]+2* stds[4]])]

results = second_order_propagation_method(x=x, f=Fun, method = 'endpoints', n_disc= 5)

CHAPTER 7

PYUNCERTAINNUMBER

Uncertain Number refers to a class of mathematical objects useful for risk analysis that generalize real numbers, intervals, probability distributions, interval bounds on probability distributions (i.e. probability boxes), and finite DempsterShafer structures.

7.1 features

- `PyUncertainNumber` is a Python package for generic computational tasks focussing on rigourou uncertainty analysis, which provides a research-grade computing environment for uncertainty characterisation, propagation, validation and uncertainty extrapolation.
- `PyUncertainNumber` also features great natural language support as such characterisatin of input uncertainty can be intuitively done by using natural language like `about 7` or simple expression like `[15 +- 10%]`, without worrying about the elicitation.
- features the save and loading of UN objects
- yields much informative results such as the combination that leads to the maximum in vertex method.

7.2 quick start

`PyUncertainNumber` can be used to easily create a `PBox` or an `Interval` object:

```
from pyuncertainnumber import UncertainNumber as UN

e = UN(
    name='elas_modulus',
    symbol='E',
    units='Pa',
    essence='pbox',
    distribution_parameters=['gaussian', ([0,12],[1,4])])
```

7.3 installation

bla bla simple bla bla

```
pip install pyuncertainnumber
```

7.4 UQ multiverse

UQ is a big world (like Marvel multiverse) consisting of abundant theories and software implementations on multiple platforms. Some notable examples include OpenCossan, UQlab in Matlab and ProbabilityBoundsAnalysis.jl in Julia, and many others of course. `PyUncertainNumber` builds upon on a few pioneering projects and will continue to be dedicated to support imprecise analysis in engineering using Python.

7.5 Acknowledgements

`PyUncertainNumber` was originally developed for use in the DAWS2 project. It has the capacity serve as a full-fleged UQ softwware to work beyond to fulfill general UQ challenges.

7.6 API references

PYTHON MODULE INDEX