PBA-for-python

Release v0.16.2

Nick Gray

Feb 02, 2024

CONTENTS

CHAPTER

ONE

INTERVALS

class pba.interval.**Interval**

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.

1.1 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.

1.2 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*.

Attributes:

`left`: The left boundary of the interval.

`right`: The right boundary of the interval.

Default values

If only 1 argument is given then the interval is assumed to be zero width around this value.

If no arguments are given then the interval is assumed to be vaccous (i.e. $[-\infty, \infty]$). This is implemented as `Interval(-np.inf,np.inf)`.

__init__(*left=None*, *right=None*)

Attributes:

`left`: The left boundary of the interval.

`right`: The right boundary of the interval.

Default values

If only 1 argument is given then the interval is assumed to be zero width around this value.

If no arguments are given then the interval is assumed to be vaccous (i.e. $[-\infty, \infty]$). This is implemented as `Interval(-np.inf,np.inf)`.

---

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`

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
```

lo()

Returns:
`self.left`

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

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:

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

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()

sample(*seed=None*, *numpy_rng: Generator | None = 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
```

1.3 Plus-Minus Intervals

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

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

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]
```

`pba.interval.`**`pm_repr`**`()`

Modifies the interval class to display the interval in [midpoint ± half-width] format.

Example:

```
>>> import pba
>>> pba.interval.pm_repr()
>>> a = pba.Interval(0,1) # defined using left and right. This cannot be overriden.
>>> b = pba.PM(0,1) # defined using midpoint and half-width
>>> print(a)
Interval [0.5 ± 0.5]
>>> print(b)
Interval [0 ± 1]
```

See also:

`lr_interval_repr()`

`pba.interval.`**`lr_repr`**`()`

Modifies the interval class to display the interval in [left, right] format.

Note: This function primarily exists to undo the effects of `pm_interval_repr()`. By default the interval class displays in this format.

Example:

```
>>> import pba
>>> pba.interval.pm_repr()
>>> a = pba.Interval(0,1) # defined using left and right, this cannot be overriden.
>>> print(a)
Interval [0.5±0.5]
>>> pba.interval.lr_repr()
>>> b = pba.PM(0,1) # defined using midpoint and half-width
>>> print(b)
Interval [-1,1]
```

CHAPTER

TWO

PROBABILITY BOXES

class pba.pbox.**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 simplest kind of p-box can be expressed mathematically as

$$\mathcal{F}(x) = [\underline{F}(x), \overline{F}(x)],\ \underline{F}(x) \geq \overline{F}(x)\ \forall x \in \mathbb{R}$$

where $\underline{F}(x)$ is the function that defines the left bound of the p-box and $\overline{F}(x)$ defines the right bound of the p-box. In PBA the left and right bounds 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, precise probability distributions can be defined in PBA 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. Resultantly, all methodology that applies to p-boxes can also be applied to probability distributions.

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.

Attention: It is usually better to define p-boxes using distributions or non-parametric methods (see). This constructor is provided for completeness and for the construction of p-boxes with precise inputs.

Parameters

left – Left bound of the p-box. Can be a list, NumPy array, Interval or numeric type. If left is None, the left bound is set to -inf.

__init__(*left=None*, *right=None*, *steps=None*, *shape=None*, *mean_left=None*, *mean_right=None*, *var_left=None*, *var_right=None*, *interpolation='linear'*)

Attention: It is usually better to define p-boxes using distributions or non-parametric methods (see). This constructor is provided for completeness and for the construction of p-boxes with precise inputs.

Parameters

left – Left bound of the p-box. Can be a list, NumPy array, Interval or numeric type. If left is None, the left bound is set to -inf.

add(*other:* Pbox | Interval | *float* | *int*, *method='f'*) → *Pbox*

Adds to Pbox to other using the defined dependency 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.

get_x()

returns the x values for plotting

get_y()

returns the y values for plotting

hi()

Returns the right-most value in the interval

imp(*other*)

Returns the imposition of self with other

lo()

Returns the left-most value in the interval

mean() → *Interval*

Returns the mean of the pbox

median() → *Interval*

Returns the median of the distribution

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

pow(*other:* Pbox | Interval | *float* | *int*, *method='f'*) → *Pbox*

Raises a p-box to the power of other using the defined dependency method

Parameters

- **other** – Pbox, Interval or numeric type
- **method** –

Returns
Pbox

Return type
Pbox

straddles(*N*, *endpoints=True*) → bool

2.1 Parameters

N
[numeric] Number to check

endpoints
[bool] Whether to include the endpoints within the check

2.2 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

truncate(*a*, *b*, *method='f'*)

Equivalent to self.min(a,method).max(b,method)

CHAPTER

THREE

DISTRIBUTIONS

3.1 Distribution based p-boxes

pba.dists.**beta**(**args*, *steps=200*)

Beta distribution

pba.dists.**lognorm**(*mean*, *var*, *steps=200*)

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>

3.2 Parameters

mean

[] mean of the lognormal distribution

var :

variance of the lognormal distribution

3.3 Returns

Pbox

pba.dists.**lognormal**(*mean*, *var*, *steps=200*)

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>

3.4 Parameters

mean
: [] mean of the lognormal distribution

var :
: variance of the lognormal distribution

3.5 Returns

Pbox

CHAPTER

FOUR

NON-PARAMETRIC P-BOXES

Functions that can be used to generate non-parametric p-boxes. These functions are used to generate p-boxes based upon the minimum, maximum, mean, median, mode, standard deviation, variance, and coefficient of variation of the variable.

pba.non_parametric.**what_I_know**(*minimum:* Interval | *float* | *int* | *None* = *None*, *maximum:* Interval | *float* | *int* | *None* = *None*, *mean:* Interval | *float* | *int* | *None* = *None*, *median:* Interval | *float* | *int* | *None* = *None*, *mode:* Interval | *float* | *int* | *None* = *None*, *std:* Interval | *float* | *int* | *None* = *None*, *var:* Interval | *float* | *int* | *None* = *None*, *cv:* Interval | *float* | *int* | *None* = *None*, *percentiles: dict[*Interval | *float* | *int]* | *None* = *None*, *debug: bool* = *False*, *steps: int* = *200*) → *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`: Mode 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: Interval(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

pba.non_parametric.**box**(*a:* Interval | *float* | *int*, *b:* Interval | *float* | *int* | *None* = *None*, *steps=200*, *shape='box'*) → *Pbox*

Returns a box shaped Pbox. This is equivalent to an Interval expressed as a Pbox.

Parameters:

`a` : Left side of box `b`: Right side of box

Returns:

`Pbox`

pba.non_parametric.**min_max**(*a:* Interval | *float* | *int*, *b:* Interval | *float* | *int* | *None* = *None*, *steps=200*, *shape='box'*) → *Pbox*

Returns a box shaped Pbox. This is equivalent to an Interval expressed as a Pbox.

Parameters:

a : Left side of box b: Right side of box

Returns:

Pbox

pba.non_parametric.**min_max_mean**(*minimum:* Interval | *float* | *int*, *maximum:* Interval | *float* | *int*, *mean:* Interval | *float* | *int*, *steps: int = 200*) → *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

pba.non_parametric.**min_max_mean_std**(*minimum:* Interval | *float* | *int*, *maximum:* Interval | *float* | *int*, *mean:* Interval | *float* | *int*, *std:* Interval | *float* | *int*, *steps: int = 200*) → *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()

pba.non_parametric.**min_max_mean_var**(*minimum:* Interval | *float* | *int*, *maximum:* Interval | *float* | *int*, *mean:* Interval | *float* | *int*, *var:* Interval | *float* | *int*, *steps: int = 200*) → *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()

pba.non_parametric.**min_max_mode**(*minimum:* Interval | *float* | *int*, *maximum:* Interval | *float* | *int*, *mode:* Interval | *float* | *int*, *steps:* *int* = *200*) → *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`

pba.non_parametric.**min_max_median**(*minimum:* Interval | *float* | *int*, *maximum:* Interval | *float* | *int*, *median:* Interval | *float* | *int*, *steps:* *int* = *200*) → *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`

pba.non_parametric.**min_max_median_is_mode**(*minimum:* Interval | *float* | *int*, *maximum:* Interval | *float* | *int*, *m:* Interval | *float* | *int*, *steps:* *int* = *200*) → *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`

pba.non_parametric.**mean_std**(*mean:* Interval | *float* | *int*, *std:* Interval | *float* | *int*, *steps=200*) → *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`

pba.non_parametric.**mean_var**(*mean:* Interval | *float* | *int*, *var:* Interval | *float* | *int*, *steps=200*) → *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`

pba.non_parametric.**pos_mean_std**(*mean:* Interval | *float* | *int*, *std:* Interval | *float* | *int*, *steps=200*) → *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`

pba.non_parametric.**symmetric_mean_std**(*mean:* Interval | *float* | *int*, *std:* Interval | *float* | *int*, *steps: int = 200*) → *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`

pba.non_parametric.**from_percentiles**(*percentiles: dict*, *steps: int = 200*) → *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: Interval(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 Interval. If value is a number, the percentile is assumed to be a point percentile. If value is an Interval, 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.

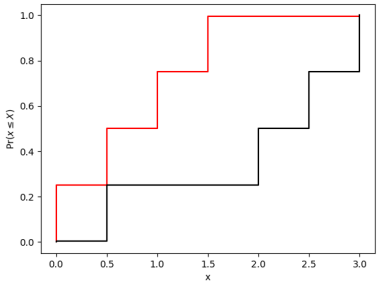
Error: `ValueError`: If any of the percentiles are not between 0 and 1.

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()
```



CHAPTER

FIVE

LOGICAL

class pba.logical.**Logical**

Bases: *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.

Attributes:

`left`: The left boundary of the interval.

`right`: The right boundary of the interval.

Default values

If only 1 argument is given then the interval is assumed to be zero width around this value.

If no arguments are given then the interval is assumed to be vaccous (i.e. $[-\infty, \infty]$). This is implemented as `Interval(-np.inf,np.inf)`.

__init__(*left: bool*, *right: bool | None = None*)

Attributes:

`left`: The left boundary of the interval.

`right`: The right boundary of the interval.

Default values

If only 1 argument is given then the interval is assumed to be zero width around this value.

If no arguments are given then the interval is assumed to be vaccous (i.e. $[-\infty, \infty]$). This is implemented as `Interval(-np.inf,np.inf)`.

---

pba.logical.**always**(*logical:* Logical | Interval | *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
```

pba.logical.**is_same_as**(*a:* Pbox | Interval, *b:* Pbox | 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
```

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
```

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
```

`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
```

(continues on next page)

(continued from previous page)

```
>>> pba.xtimes(c < b)
True
```

CHAPTER

SIX

CONFIDENCE BOXES

class pba.cbox.**Cbox**

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.

Args:

Pbox (_type_): _description_

> **Attention:** It is usually better to define p-boxes using distributions or non-parametric methods (see). This constructor is provided for completeness and for the construction of p-boxes with precise inputs.

Parameters

left – Left bound of the p-box. Can be a list, NumPy array, Interval or numeric type. If left is None, the left bound is set to -inf.

__init__(**args*, ***kwargs*)

> **Attention:** It is usually better to define p-boxes using distributions or non-parametric methods (see). This constructor is provided for completeness and for the construction of p-boxes with precise inputs.

Parameters

left – Left bound of the p-box. Can be a list, NumPy array, Interval or numeric type. If left is None, the left bound is set to -inf.

add(*other*, *method='f'*)

Adds to Pbox to other using the defined dependency method

CHAPTER

SEVEN

USEFUL FUNCTIONS

Here are some useful functions to use with PBA objects.

pba.core.**env**(**args*)

> **Warning:** Deprecated function, use envelope() instead.

pba.core.**envelope**(**args:* Interval | Pbox | *float*) → *Interval* | *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.

pba.core.**mean**(**args: 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)
```

---

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

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

pba.core.**sum**(**args: 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.

CHAPTER

EIGHT

DEPENDENCY ARITHMETIC

In PBA for Python, there are 4 default dependency types:

Frechet `f`

Indepedence `i`

Perfect `p`

Opposite `o`

PYTHON MODULE INDEX

p

INDEX

N

O

P

R

S

T

W

X