affapy

Release 0.1

Q. DESCHAMPS, F. GUILY, T. MICHEL, R. ZENG

May 28, 2020

CONTENTS:

CHAPTER

ONE

PRESENTATION

Affapy is a Python library for multiprecision Affine Arithmetic. *Affapy* can be used to perform operations with affine forms and intervals.

These documentation is a listing of all methods of the two classes:

- **Affine**: affine arithmetic (AA)
- **Interval**: interval arithmetic (IA)

There are explanations about the implementation and examples.

1.1 Installation

You can install latest release of the *affapy* library with pip3:

```
pip3 install affapy
```

The library uses *mpmath* for multiprecision calculations.

You can also install *numpy* and *matplotlib* to launch the examples 4 and 5.

1.2 Basic usage

1.2.1 Affine Arithmetic

With *affapy*, you can create affine forms and perform operations:

```
from affapy.aa import Affine

# Init
x = Affine([1, 2])
y = Affine([3, 4])

# Get the interval
x.interval
y.interval

# Basic operations
x + y
x + 5
x - y
```

(continues on next page)

(continued from previous page)

```
x - 5
-x

# Advanced operations
x * y
x * 2
x / y
2 / x
x ** y
x ** 3

# Functions
abs(x)
x.sqrt()
x.exp()
x.log()

# Trigonometry
x.sin()
x.cos()
x.tan()
x.cotan()

# Hyperbolic functions
x.cosh()
x.sinh()
x.tanh()

# Comparisons
x == y
x != y
x in y
```

1.2.2 Interval Arithmetic

You can also create intervals and perform operations:

```
from affapy.ia import Interval

# Init
x = Interval(1, 2)
y = Interval(3, 4)
```

The operators and the functions have the same syntax than Affine Arithmetic. Nevertheless, there are other comparison operators for intervals:

```
# Comparisons
x == y
x != y
x in y
x >= y
x > y
x <= y
x < y
```

1.2.3 Precicion context

You can set the precision of your calculations using the **precision** module:

```
from affapy.precision import precision

with precision(dps=30):
    x + y

@precision(dps=30)
def eval_fct(x, y):
    return x + y
```

1.3 AffApy

The *affapy* library contains 4 modules:

1.3.1 Affine Arithmetic (AA)

This module can create affine forms and perform operations.

Affine Arithmetic (AA) has been developed to overcome the error explosion problem of standard Interval Arithmetic (IA). This method represents a quantity x as an affine form $\hat{x}$, which is a first degree polynomial:

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \epsilon_i$$

The coefficients x_i are finite floating-point numbers: they are called the *partial deviations*.

The coefficient x_0 is the *central value* of the affine form $\hat{x}$.

The ϵ_i coefficients are symbolic real values called *noise symbols*. Their values are unknown between -1 and 1.

This representation enables a better tracking of the different quantities inside the affine form.

For example, the quantity $[0, 10]$ can be represented as the following affine form:

$$A = [0, 10] = 5 + 5\epsilon_1$$

where

$$x_0 = 1, x_1 = 5$$

But we could also represent it like this:

$$B = [0, 10] = 5 + 3\epsilon_1 + 2\epsilon_2$$

where

$$x_0 = 5, x_1 = 3, x_2 = 2$$

Both forms represent the same quantity but they are handling differently the storage of internal quantities. They will behave differently during operation:

$$A - A = 0$$

whereas

$$A - B = 0 + 2\epsilon_1 - 2\epsilon_2 = [0, 4]$$

The second example illustrates this behaviour. Even though A and B represent the same quantity, they manage their quantity differently. Therefore, they are not equal.

class aa.**Affine**(*interval=None*, *x0=None*, *xi=None*)

Bases: object

Representation of an affine form. An instance of the class **Affine** is composed of three fields:

- **interval**: the interval associated to the affine form
- **x0**: the center
- **xi**: the dictionnary of noise symbols

__init__(*interval=None*, *x0=None*, *xi=None*)

Create an affine form. There are two different ways:

```
x1 = Affine(interval=[inf, sup])
x2 = Affine(x0=0, xi={})
```

If no arguments, x0=0 and xi={}.

The first method is easier to use. To convert an interval $[a, b]$ into an affine form, there is the formula:

$$\hat{x} = x_0 + x_k\epsilon_k$$

with:

$$x_0 = \frac{a+b}{2}, x_k = \frac{a-b}{2}$$

To convert an affine form into an interval X:

$$X = [x_0 + rad(x), x_0 - rad(x)]$$

with:

$$rad(x) = \sum_{i=1}^{n} |x_i|$$

Parameters

- **interval** (*list or tuple with length 2 or Interval*) – the interval
- **x0** (*int or float or mpf*) – the center
- **xi** (*dict of mpf values*) – noise symbols

Returns affine form

Return type *Affine*

Raises ***affapyError*** – interval must be list, tuple or Interval

Examples

```
>>> from affapy.aa import Affine
>>> Affine([1, 3])
Affine(2.0, {5: mpf('-1.0')})
mpf('1.0')
>>> print(Affine(x0=1, xi={1:2, 2:3}))
1.0 + 2.0e1 + 3.0e2
```

property interval
Return interval associated to the affine form.

property x0
Return the center x0.

property xi
Return the dictionnary of noice symbols xi.

static _getNewXi()
Get a new noise symbol.

rad()
Return the radius of an affine form:

$$rad(x) = \sum_{i=1}^{n} |x_i|$$

Parameters **self** (Affine) – operand

Returns sum of abs(xi)

Return type mpf

Examples

```
>>> x = Affine([1, 3])
>>> x.rad()
mpf('1.0')
```

__neg__()
Operator - (unary)

Return the additive inverse of an affine form:

$$-\hat{x} = -x_0 + \sum_{i=1}^{n} -x_i \epsilon_i$$

Parameters **self** (Affine) – operand

Returns -self

Return type *Affine*

Examples

```
>>> print(-Affine([1, 2]))
-1.5 + 0.5e1
```

__add__(*other*)

Operator +

Add two affine forms:

$$\hat{x} + \hat{y} = (x_0 + y_0) + \sum_{i=1}^{n} (x_i + y_i)\epsilon_i$$

Or add an affine form and an integer or float or mpf:

$$\hat{x} + y = (x_0 + y) + \sum_{i=1}^{n} x_i\epsilon_i$$

Parameters

- **self** (*Affine*) – first operand
- **other** (*Affine or int or float or mpf*) – second operand

Returns self + other

Return type *Affine*

Raises ***affapyError*** – other must be Affine, int, float, mpf

Examples

```
>>> print(Affine([0, 1]) + Affine([3, 4]))
4.0 + -0.5e1 + -0.5e2
>>> print(Affine([1, 2]) + 3)
4.5 + -0.5e1
```

__radd__(*other*)

Reverse operator +

Add two affine forms or an affine form and an integer or float or mpf. See the add operator for more details.

Parameters

- **self** (*Affine*) – second operand
- **other** (*Affine or int or float or mpf*) – first operand

Returns other + self

Return type *Affine*

Raises ***affapyError*** – other must be Affine, int, float, mpf

Examples

```
>>> print(1 + Affine([1, 2]))
2.5 + -0.5e3
```

__sub__(*other*)

Operator -

Subtract two affine forms:

$$\hat{x} - \hat{y} = (x_0 - y_0) + \sum_{i=1}^{n} (x_i - y_i)\epsilon_i$$

Or subtract an affine form and an integer or float or mpf:

$$\hat{x} - y = (x_0 - y) + \sum_{i=1}^{n} x_i\epsilon_i$$

Parameters

- **self** (*Affine*) – first operand
- **other** (*Affine or int or float or mpf*) – second operand

Returns self - other

Return type *Affine*

Raises ***affapyError*** – other must be Affine, int, float, mpf

Examples

```
>>> print(Affine([0, 1]) - Affine([3, 4]))
-3.0 + -0.5e4 + 0.5e5
>>> print(Affine([1, 2]) + 3)
-1.5 + -0.5e6
```

__rsub__(*other*)

Reverse operator -

Subtract two affine forms or an integer or float or mpf and an affine form. See the sub operator for more details.

Parameters

- **self** (*Affine*) – second operand
- **other** (*Affine or int or float or mpf*) – first operand

Returns other - self

Return type *Affine*

Raises ***affapyError*** – other must be Affine, int, float, mpf

Examples

```
>>> print(3 - Affine([1, 2]))
1.5 + 0.5e1
```

__mul__(*other*)

Operator *

Multiply two affine forms:

$$\hat{x}\hat{y} = x_0 y_0 + \sum_{i=1}^{n}(x_0 y_i + y_0 x_i)\epsilon_i + rad(x)rad(y)\epsilon_k$$

k is a new noise symbol. Or multiply an affine form and integer or float or mpf:

$$\hat{x}y = x_0 y + \sum_{i=1}^{n} x_i y \epsilon_i$$

Parameters

- **self** (*Affine*) – first operand
- **other** (*Affine or int or float or mpf*) – second operand

Returns self * other

Return type *Affine*

Raises ***affapyError*** – other must be Affine, int, float, mpf

Examples

```
>>> print(Affine([1, 2]) * Affine([3, 4]))
5.25 + -1.75e2 + -0.75e3 + 0.25e4
```

__rmul__(*other*)

Reverse operator *

Multiply two affine forms or an integer or float or mpf and an affine form. See the mul operator for more details.

Parameters

- **self** (*Affine*) – second operand
- **other** (*Affine or int or float or mpf*) – first operand

Returns other * self

Return type *Affine*

Raises ***affapyError*** – other must be Affine, int, float, mpf

_affineConstructor(*alpha*, *dzeta*, *delta*)

Affine constructor

Return the affine form for non-affine operations:

$$\hat{\chi} = (\alpha x_0 + \zeta) + \sum_{i=1}^{n} \alpha x_i \epsilon_i + \delta\epsilon_k$$

k is a new noise symbol.

Parameters

- **alpha** (*mpmath.mpf*) –
- **dzeta** (*mpmath.mpf*) –
- **delta** (*mpmath.mpf*) –

Returns construction of an affine form

Return type *Affine*

inv()

Inverse

Return the inverse of an affine form. It uses the affine constructor with:

$$\alpha = -\frac{1}{b^2}$$

$$\zeta = abs(mid(i)),$$

$$\delta = radius(i)$$

with:

$$i = [\frac{1}{a} - \alpha a, \frac{2}{b}]$$

$$a = min(|inf|, |sup|)$$

$$b = max(|inf|, |sup|)$$

where $[inf, sup]$ is the interval associated to the affine form in argument.

Parameters **self** – operand

Returns 1 / self

Return type *Affine*

Raises ***affapyError*** – the interval associated to the affine form contains 0

__truediv__(*other*)

Operator /

Divide two affine forms or an integer or float or mpf and an affine form. It uses the identity:

$$\frac{x}{y} = x \times \frac{1}{y}$$

Parameters

- **self** (Affine) – first operand
- **other** (*Affine or int or float or mpf*) – second operand

Returns self / other

Return type *Affine*

Raises ***affapyError*** – other must be Affine, int, float, mpf

Examples

```
>>> print(Affine([1, 2]) / Affine([3, 4]))
0.4375 + -0.145833333333333e11 + 0.046875e12 +
0.0156249999999999e13 + 0.0208333333333333e14
```

__rtruediv__(*other*)

Reverse operator /

Divide two affine forms or an affine form and an integer or float or mpf. See the truediv operator for more details.

Parameters

- **self** (*Affine*) – second operand
- **other** (*Affine or int or float or mpf*) – first operand

Returns other / self

Return type *Affine*

Raises ***affapyError*** – other must be Affine, int, float, mpf

Examples

```
>>> print(2 / Affine([1, 2]))
1.5 + 0.25e15 + 0.25e16
```

sqr()

Return the square of an affine form. It uses the identity:

$$x^2 = x \times x$$

Parameters **self** (*Affine*) – operand

Returns self ** 2

Return type *Affine*

__pow__(*n*)

Operator **

Return the power of an affine form with another affine form or an integer. With an affine, it uses the identity:

$$x^n = exp(n \times log(x))$$

Parameters

- **self** (*Affine*) – first operand
- **n** (*Affine or int*) – second operand (exponent)

Returns self ** n

Return type *Affine*

Raises ***affapyError*** – type error: n must be Affine or int

Examples

```
>>> print(Affine([1, 2])**3)
3.375 + -3.375e17 + 0.375e18 + 0.875e19
```

__abs__()

Return the absolute value of an affine form. Three possibilities:

1. If $x < 0$:

$$|\hat{x}| = -\hat{x}$$

2. If $x > 0$:

$$|\hat{x}| = \hat{x}$$

3 If x straddles 0:

$$|\hat{x}| = \frac{|x_0|}{2} + \sum_{i=1}^{n} \frac{x_i \epsilon_i}{2}$$

Parameters **self** (Affine) – operand

Returns abs(self)

Return type *Affine*

Examples

```
>>> print(abs(Affine([1, 2])))
1.5 + -0.5e24
>>> print(abs(Affine([-2, -1])))
1.5 + 0.5e25
```

sqrt()

Function sqrt

Return the square root of an affine form. We consider the interval $[a, b]$ associated to the affine form. It uses the affine constructor with:

$$\alpha = \frac{1}{\sqrt{b} + \sqrt{a}}$$

$$\zeta = \frac{\sqrt{a} + \sqrt{b}}{8} + \frac{1}{2} \frac{\sqrt{a}\sqrt{b}}{\sqrt{a} + \sqrt{b}}$$

$$\delta = \frac{1}{8} \frac{(\sqrt{b} - \sqrt{a})^2}{\sqrt{a} + \sqrt{b}}$$

Parameters **self** (Affine) – operand

Returns sqrt(self)

Return type *Affine*

Raises ***affapyError*** – the interval associated to the affine form must be >=0

Examples

```
>>> print(Affine([1, 2]).sqrt())
1.21599025766973 + -0.207106781186548e27 + 0.0088834764831844 1e28
```

exp()

Function exp

Return the exponential of an affine form. We consider the interval $[a, b]$ associated to the affine form. It uses the affine constructor with:

$$\alpha = \frac{exp(b) - exp(a)}{b - a}$$

$$\zeta = \alpha \times (1 - log(\alpha))$$

$$\delta = \frac{\alpha \times (log(\alpha) - 1 - a)) + exp(a)}{2}$$

Parameters **self** (Affine) – operand

Returns exp(self)

Return type *Affine*

Examples

```
>>> print(Affine([1, 2]).exp())
4.47775520281461 + -2.3353871352358e29 + 4.95873115091173e30
```

log()

Function log

Return the logarithm of an affine form. We consider the interval $[a, b]$ associated to the affine form. It uses the affine constructor with:

$$\alpha = \frac{log(b) - log(a)}{b - a}$$

$$\zeta = \frac{-\alpha x_s}{\frac{log(x_s) + y_s}{2}}$$

$$\delta = \frac{log(x_s) - y_s}{2}$$

with:

$$x_s = \frac{1}{\alpha}$$

$$y_s = \alpha(x_s - a) + log(a)$$

Parameters **self** (Affine) – operand

Returns log(self)

Return type *Affine*

Raises ***affapyError*** – the interval associated to the affine form must be > 0

Examples

```
>>> print(Affine([1, 2]).log())
-1.93043330907435 + -0.346573590279973e31 + 0.0298300505708048e32
```

sin(*npts=8*)

Function sin

Return the sinus of an affine form. It uses the least squares and the affine constructor. The argument npts is the number of points for the linear regression approximation.

Parameters

- **self** (Affine) – operand
- **npts** (*int*) – number of points (default: 8)

Returns sin(self)

Return type *Affine*

Examples

```
>>> print(Affine([1, 2]).sin())
0.944892253579443 + -0.0342679845626557e33 + 0.0698628113164167e34
```

cos()

Function cos

Return the cosinus of an affine form. It uses the identity:

$$cos(x) = sin\left(x + \frac{\pi}{2}\right)$$

Parameters **self** (Affine) – operand

Returns cos(self)

Return type *Affine*

tan()

Function tan

Return the tangent of an affine form. It uses the identity:

$$tan(x) = \frac{sin(x)}{cos(x)}$$

Parameters **self** (Affine) – operand

Returns tan(self)

Return type *Affine*

cotan()

Function cotan

Return the cotangent of an affine form. It uses the identity:

$$cotan(x) = \frac{cos(x)}{sin(x)}$$

Parameters **self** (Affine) – operand

Returns cotan(self)

Return type *Affine*

cosh()

Function cosh

Return the hyperbolic cosine of an affine form. It uses the identity:

$$cosh(x) = \frac{exp(x) + exp(-x)}{2}$$

Parameters **self** (Affine) – operand

Returns cosh(self)

Return type *Affine*

sinh()

Function sinh

Return the hyperbolic sine of an affine form. It uses the identity:

$$sinh(x) = \frac{exp(x) - exp(-x)}{2}$$

Parameters **self** (Affine) – operand

Returns sinh(self)

Return type *Affine*

tanh()

Function tanh

Return the hyperbolic tangeant of an affine form. It uses the identity:

$$tanh(x) = \frac{sinh(x)}{cosh(x)}$$

Parameters **self** (Affine) – operand

Returns tanh(self)

Return type *Affine*

__eq__(*other*)

Operator ==

Compare two affine forms.

Parameters

- **self** (Affine) – first operand
- **other** (Affine) – second operand

Returns self == other

Return type bool

Raises ***affapyError*** – other must be Affine

__ne__(*other*)

Operator !=

Negative comparison of two affine forms.

Parameters

- **self** (Affine) – first operand
- **other** (Affine) – second operand

Returns self != other

Return type bool

Raises ***affapyError*** – other must be Affine

__contains__(*other*)

Operator in

Return True if the interval of self is in the interval of other.

Parameters

- **self** (Affine) – first operand
- **other** (Affine) – second operand

Returns self in other

Return type bool

Raises ***affapyError*** – other must be Affine, Interval, int, float, mpf

straddles_zero()

Return True if the affine form straddles 0, False if not.

Parameters **self** (Affine) – operand

Returns 0 in self

Return type bool

strictly_neg()

Return True if the affine is strictly negative, False if not.

Parameters **self** (Affine) – operand

Returns self < 0

Return type bool

__str__()

String format

Make the string format.

Parameters **self** (Affine) – arg

Returns sum of noise symbols

Return type string

Examples

```
>>> print(Affine([1, 2]))
1.5 - 0.5*e1
```

__repr__()

Repr format

Make the repr format.

Parameters **self** (Affine) – arg

Returns format

Return type string

copy()

Copy an affine form.

Parameters **self** (Affine) – arg

Returns self copy

Return type *Affine*

convert()

Convert an affine form to an interval representation:

$$X = [x_0 + rad(x), x_0 - rad(x)]$$

with:

$$rad(x) = \sum_{i=1}^{n} |x_i|$$

Parameters **self** (Affine) – arg

Returns interval associated to the affine form

Return type *Interval*

1.3.2 Interval Arithmetic (IA)

This module can create intervals and perform operations.

In order to bound rounding errors when performing floating point arithmetic, we can use Interval Arithmetic (IA) to keep track of rounding errors.

After a series of operations using basic operators like +, -, * and / we end of with an interval instead of an approximation of the result.

The interval width represents the uncertainty of the result but we would know for sure that the correct result will be within this interval.

An interval is presented by two number representing the lower and upper range of the interval:

$$[a, b]$$

where $a \leq b$.

class ia.**Interval**(*inf*, *sup*)

Bases: object

Representation of an interval. An instance of the class **Interval** is composed of two fields:

- **inf**: the infimum
- **sup**: the supremum

__init__(*inf*, *sup*)

Create an interval. It is composed of two fields : the infimum and the supremum. If $inf > sup$, then the init function reorganize the two values.

Parameters

- **inf** (*int or float or string*) – infimum
- **sup** (*int or float or string*) – supremum

Returns interval

Return type *Interval*

Examples

```
>>> from affapy.ia import Interval
>>> x = Interval(1, 2)
>>> print(x)
[1.0, 2.0]
```

property inf

Return the inf.

property sup

Return the sup.

width()

Width

Return the width of an interval:

$$width([a, b]) = b - a$$

Parameters **self** (Interval) – arg

Returns sup - inf

Return type mpf

Examples

```
>>> Interval(1, 2).width()
mpf('1.0')
```

mid()

Middle

Return the middle of an interval:

$$middle([a, b]) = \frac{a + b}{2}$$

Parameters **self** (Interval) – arg

Returns (inf + sup) / 2

Return type mpf

Examples

```
>>> Interval(1, 2).mid()
mpf('1.5')
```

radius()

Radius

Return the radius of an interval:

$$radius([a, b]) = \frac{width([a, b])}{2}$$

Parameters **self** (Interval) – arg

Returns width / 2

Return type mpf

Examples

```
>>> Interval(1, 2).radius()
mpf('0.5')
```

__neg__()

Operator - (unary)

Return the additive inverse of an interval:

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

Parameters **self** (Interval) – operand

Returns -self

Return type *Interval*

Examples

```
>>> -Interval(1, 2)
Interval(-2.0, -1.0)
```

__add__(*other*)

Operator +

Add two intervals:

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

Or add an interval and an integer or float or mpf:

$$[a, b] + k = [a + k, b + k]$$

Parameters

- **self** (Interval) – first operand
- **other** (*Interval or int or float or mpf*) – second operand

Returns self + other

Return type *Interval*

Raises ***affapyError*** – other must be Interval, int, float, mpf

Examples

```
>>> Interval(1, 2) + Interval(3, 4)
Interval(4.0, 6.0)
>>> Interval(1, 2) + 2
Interval(3.0, 4.0)
```

__radd__(*other*)

Reverse operator +

Add two intervals or an interval and an integer or float or mpf. See the add operator for more details.

Parameters

- **self** (Interval) – second operand
- **other** (*Interval or int or float or mpf*) – first operand

Returns other + self

Return type *Interval*

Raises ***affapyError*** – other must be Interval, int, float, mpf

Examples

```
>>> 2 + Interval(1, 2)
Interval(3.0, 4.0)
```

__sub__(*other*)

Operator -

Subtract two intervals:

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

Or subtract an interval and an integer or float or mpf:

$$[a, b] - k = [a - k, b - k]$$

Parameters

- **self** (Interval) – first operand
- **other** (*Interval or int or float or mpf*) – second operand

Returns self - other

Return type *Interval*

Raises ***affapyError*** – other must be Interval, int, float, mpf

Examples

```
>>> Interval(1, 2) - Interval(3, 4)
Interval(-3.0, -1.0)
>>> Interval(1, 2) - 3
Interval(-2.0, -1.0)
```

__rsub__(*other*)

Reverse operator -

Subtract two intervals or an interval and an integer or float or mpf. See the sub operator for more details.

Parameters

- **self** (Interval) – second operand
- **other** (*Interval or int or float or mpf*) – first operand

Returns other - self

Return type *Interval*

Raises ***affapyError*** – other must be Interval, int, float, mpf

Examples

```
>>> 1 - Interval(2, 3)
Interval(-2.0, -1.0)
```

__mul__(*other*)

Operator *

Multiply two intervals:

$$[a,b] \times [c,d] = [min\{a \times c, a \ \times d, b \ \times c, b \times d\}, max\{a \times c, a \times d, b \times c, b \times d\}]$$

Or multiply an interval and an integer or float or mpf:

$$[a,b] \times k = [a \times k, b \times k]$$

Parameters

- **self** (Interval) – first operand
- **other** (*Interval or int or float or mpf*) – second operand

Returns self * other

Return type *Interval*

Raises ***affapyError*** – other must be Interval, int, float, mpf

Examples

```
>>> Interval(1, 2) * Interval(3, 4)
Interval(3.0, 8.0)
>>> Interval(1, 2) * 3
Interval(3.0, 6.0)
```

__rmul__(*other*)
Reverse operator *

Multiply two intervals or an interval and an integer or float or mpf. See the mul operator for more details.

Parameters

- **self** (Interval) – second operand
- **other** (*Interval or int or float or mpf*) – first operand

Returns other * self

Return type *Interval*

Raises ***affapyError*** – other must be Interval, int, float, mpf

__truediv__(*other*)
Operator /

Divide two intervals:

$$[a, b]/[c, d] = [a, b] \times [1/d, 1/c]$$

or an interval and an integer or float or mpf:

$$[a, b]/k = \frac{1}{k} \times [a, b]$$

It is possible only if other does not contains 0.

Parameters

- **self** (Interval) – first operand
- **other** (Interval) – second operand

Returns self / other

Return type *Interval*

Raises

- ***affapyError*** – division by 0
- ***affapyError*** – other must be Interval, int, float, mpf

Examples

```
>>> Interval(1, 2) / Interval(3, 4)
Interval(0.25, 0.666666666666667)
>>> Interval(1, 2) / 2
Interval(0.5, 1.0)
>>> Interval(1, 2) / Interval(-1, 1)
...
affapy.affapyError.affapyError: division by 0
```

__pow__(*n*)

Operator **

Return the power of an interval with another interval or an integer. With an interval, it uses the identity:

$$x^n = exp(n \times log(x))$$

Parameters

- **self** (Interval) – first operand
- **n** (*Interval or int*) – second operand (exponent)

Returns self ** n

Return type *Interval*

Raises ***affapyError*** – type error: n must be Interval or int

Examples

```
>>> Interval(1, 2) ** Interval(3, 4)
Interval(1.0, 16.0)
>>> Interval(1, 2) ** 3
Interval(1.0, 8.0)
```

__floor__()

Function floor

Return the floor of an interval:

$$floor([a, b]) = [floor(a), floor(b)]$$

You need to import floor from the math library.

Parameters **self** (Interval) – arg

Returns [floor(inf), floor(sup)]

Return type *Interval*

Examples

```
>>> from math import floor
>>> floor(Interval(1.4, 2.5))
Interval(1.0, 2.0)
```

__ceil__()

Function ceil

Return the ceil of an interval:

$$ceil([a, b]) = [ceil(a), ceil(b)]$$

You need to import ceil from the math library.

Parameters **self** (Interval) – arg

Returns [ceil(inf), ceil(sup)]

Return type *Interval*

Examples

```
>>> from math import ceil
>>> ceil(Interval(1.4, 2.5))
Interval(2.0, 3.0)
```

__abs__()

Function abs

Return the absolute value of an interval. Three possibilities:

1. If $[a, b] < 0$:

$$abs([a, b]) = [-b, -a]$$

2. If $[a, b] > 0$:

$$abs([a, b]) = [a, b]$$

3. If $0 \in [a, b]$:

$$abs([a, b]) = [0, max\{abs(a), abs(b)\}]$$

Parameters **self** (Interval) – operand

Returns abs(self)

Return type *Interval*

Examples

```
>>> abs(Interval(-2, -1))
Interval(1.0, 2.0)
>>> abs(Interval(1, 2))
Interval(1.0, 2.0)
>>> abs(Interval(-2, 1))
Interval(0.0, 2.0)
```

sqrt()

Function sqrt

Return the square root of an interval:

$$\sqrt{[a, b]} = [\sqrt{a}, \sqrt{b}]$$

It is possible only if $a \geq 0$.

Parameters **self** (Interval) – operand

Returns sqrt(self)

Return type *Interval*

Raises ***affapyError*** – inf must be >= 0

Examples

```
>>> Interval(1, 2).sqrt()
Interval(1.0, 1.4142135623731)
>>> Interval(-1, 2).sqrt()
...
affapy.affapyError.affapyError: inf must be >= 0
```

exp()

Function exp

Return the exponential of an interval:

$$exp([a, b]) = [exp(a), exp(b)]$$

Parameters **self** (Interval) – operand

Returns exp(self)

Return type *Interval*

Examples

```
>>> Interval(1, 2).exp()
Interval(2.71828182845905, 7.38905609893065)
```

log()

Function log

Return the logarithm of an interval:

$$log([a, b]) = [log(a), log(b)]$$

It is possible only if $a > 0$.

Parameters **self** (Interval) – operand

Returns log(self)

Return type *Interval*

Raises ***affapyError*** – inf must be > 0

Examples

```
>>> Interval(1, 2).log()
Interval(0.0, 0.693147180559945)
>>> Interval(-1, 2).log()
...
affapy.affapyError.affapyError: inf must be > 0
```

minTrigo()

Return the minimal 2π periodic interval of an interval.

Parameters **self** (Interval) – operand

Returns minimal 2π periodic interval

Return type *Interval*

cos()

Function cos

Return the cosinus of an interval. It considers the minimal 2π periodic interval $[a, b]$ of the interval x. Then:

1. If $a \leq \pi$:

- if $b \leq \pi$:

$$cos(x) = [cos(b), cos(a)]$$

- if $\pi < b \leq 2\pi$:

$$cos(x) = [-1, max(cos(a), cos(b))]$$

- else:

$$cos(x) = [-1, 1]$$

2. If $\pi < a \leq 2\pi$:

- if $b \leq 2\pi$:

$$cos(x) = [cos(a), cos(b)]$$

- if $2\pi < b \leq 3\pi$:

$$cos(x) = [min(cos(a), cos(b)), 1]$$

- else:

$$cos(x) = [-1, 1]$$

Parameters **self** (Interval) – operand

Returns cos(self)

Return type *Interval*

Examples

```
>>> Interval(1, 2).cos()
Interval(-0.416146836547142, 0.54030230586814)
```

sin()

Function sin

Return the sinus of an interval. It uses the identity:

$$sin(x) = cos\left(\frac{\pi}{2} - x\right)$$

Parameters **self** (Interval) – operand

Returns sin(self)

Return type *Interval*

tan()

Function tan

Return the tangent of an interval. It uses the identity:

$$tan(x) = \frac{sin(x)}{cos(x)}$$

Parameters **self** (Interval) – operand

Returns tan(self)

Return type *Interval*

cotan()

Function cotan

Return the cotangent of an interval. It uses the identity:

$$cotan(x) = \frac{cos(x)}{sin(x)}$$

Parameters **self** (Interval) – operand

Returns cotan(self)

Return type *Interval*

cosh()

Function cosh

Return the hyperbolic cosine of an interval. It uses the identity:

$$cosh(x) = \frac{exp(x) + exp(-x)}{2}$$

Parameters **self** (Interval) – operand

Returns cosh(self)

Return type *Interval*

sinh()

Function sinh

Return the hyperbolic sine of an interval. It uses the identity:

$$sinh(x) = \frac{exp(x) - exp(-x)}{2}$$

Parameters **self** (Interval) – operand

Returns sinh(self)

Return type *Interval*

tanh()

Function tanh

Return the hyperbolic tangeant of an interval. It uses the identity:

$$tanh(x) = \frac{sinh(x)}{cosh(x)}$$

Parameters **self** (Interval) – operand

Returns tanh(self)

Return type *Interval*

__eq__(*other*)

Operator ==

Compare two intervals.

Parameters

- **self** (Interval) – first operand
- **other** (Interval) – second operand

Returns self == other

Return type bool

Raises ***affapyError*** – other must be Interval

Examples

```
>>> Interval(1, 2) == Interval(1, 2)
True
>>> Interval(1, 2) == Interval(1, 3)
False
```

__ne__(*other*)

Operator !=

Negative comparison of two intervals.

Parameters

- **self** (Interval) – first operand
- **other** (Interval) – second operand

Returns self != other

Return type bool

Raises ***affapyError*** – other must be Interval

Examples

```
>>> Interval(1, 2) != Interval(1, 2)
False
>>> Interval(1, 2) != Interval(1, 3)
True
```

__ge__(*other*)

Operator >=

Greater or equal comparison between two intervals or with int, float or mpf.

Parameters

- **self** (Interval) – first operand
- **other** (*Interval or int or float or mpf*) – second operand

Returns self >= other

Return type bool

Raises ***affapyError*** – other must be Interval, int, float, mpf

Examples

```
>>> Interval(1, 2) >= Interval(0, 1)
True
>>> Interval(1, 2) >= Interval(0, 2)
False
```

__gt__(*other*)

Operator >

Greater comparison between two intervals or with int, float or mpf.

Parameters

- **self** (Interval) – first operand
- **other** (*Interval or int or float or mpf*) – second operand

Returns self > other

Return type bool

Raises ***affapyError*** – other must be Interval, int, float, mpf

Examples

```
>>> Interval(1, 2) > Interval(0, 1)
False
```

__le__(*other*)

Operator <=

Lesser or equal comparison between two intervals or with int, float or mpf.

Parameters

- **self** (Interval) – first operand
- **other** (*Interval or int or float or mpf*) – second operand

Returns self <= other

Return type bool

Raises ***affapyError*** – other must be Interval, int, float, mpf

Examples

```
>>> Interval(1, 2) <= Interval(2, 3)
True
>>> Interval(1, 2) <= Interval(1, 3)
False
```

__lt__(*other*)

Operator <

Lesser comparison between two Intervals or with int, float or mpf.

Parameters

- **self** (*Interval*) – first operand
- **other** (*Interval or int or float or mpf*) – second operand

Returns self < other

Return type bool

Raises ***affapyError*** – other must be Interval, int, float, mpf

Examples

```
>>> Interval(1, 2) <= Interval(2, 3)
False
```

__contains__(*other*)

Operator in

Return True if other, who is interval, or int, or float or mpf, is in self, who is an interval:

$$x \in [a, b]$$

Parameters

- **self** (*Interval*) – second operand
- **other** (*Interval or Affine or int or float or mpf*) – first operand

Returns other in self

Return type bool

Raises ***affapyError*** – if other is not Interval, int, float, Affine, mpf

Examples

```
>>> Interval(1, 2) in Interval(1, 3)
True
>>> Interval(1, 4) in Interval(1, 3)
False
```

straddles_zero()

Return True if the interval straddles 0, False if not.

Parameters **self** (*Interval*) – operand

Returns self straddles 0

Return type bool

__str__()

String format

Make the string format.

Parameters **self** (Interval) – arg

Returns the infimum and the supremum

Return type string

Examples

```
>>> print(Interval(1, 2)
[1, 2]
```

__repr__()

Repr format

Make the repr format.

Parameters **self** (Interval) – arg

Returns format

Return type string

copy()

Copy an interval.

Parameters **self** (Interval) – arg

Returns self copy

Return type *Interval*

convert()

Convert an interval $[a, b]$ to an affine form:

$$\hat{x} = x_0 + x_k \epsilon_k$$

with:

$$x_0 = \frac{a+b}{2}, x_k = \frac{a-b}{2}$$

Parameters **self** (Interval) – operand

Returns affine form associated to the interval

Return type *Affine*

1.3.3 Precision context

This module manage the context precision of calculations using *affapy*.

Indeed, you can choice the precision using the **precision** class. You can set different precision contexts between functions using the precision decorator or use the class using the *with* statement.

This class changes the precision context of *mpmath*.

You can see **exPrecision1** and **exPrecision2** to see how it works.

class precision.**precision**(*dps: int = None*, *prec: int = None*)

Bases: contextlib.ContextDecorator

Manage precision for *affapy* library. You can use it:

- As decorator of a function
- Using the *with* statement

It contains four fields:

- **dps**: decimal precision (decimals number)
- **prec**: binary precision (bits number)
- **old_dps**: decimal precision before entry to the context
- **old_prec**: binary precision before entry to the context

Example:

```
from affapy.precision import precision

with precision(dps=30):
    x + y

@precision(dps=30)
def eval_fct(x, y):
    return x + y
```

__init__(*dps: int = None*, *prec: int = None*)

Init the context manager for precision. You need to mention dps or prec. Both is useless.

Parameters

- **dps** (*int*) – decimal precision of mpmath
- **prec** (*int*) – binary precision of mpmath

Raises ***affapyError*** – Invalid value for precision

property dps

Get decimal precision.

property prec

Get binary precision.

property old_dps

Get old decimal precision.

property old_prec

Get old binary precision.

static set_dps(*dps: int*)

Set decimal precision outside precision class.

Parameters **dps** (*int*) – decimal precision

static set_prec(*prec: int*)
Set binary precision outside precision class.

Parameters **prec** (*int*) – binary precision

__enter__()
Set the *mpmath* precision for a portion of code.

Raises ***affapyError*** – No precision mentioned

__exit__(*exc_type*, *exc_val*, *exc_tb*)
Reset the *mpmath* precision to its last value.

Raises ***affapyError*** – No precision saved

1.3.4 Errors

This module manage the errors of the *affapy* library.

exception error.**affapyError**
Bases: Exception

Manage exception errors.

exception error.**affapyWarning**
Bases: UserWarning

Manage warnings.

1.4 Examples

This is the list of the different examples that shows how to use the *affapy* library. The examples 1 to 5 are inspired by the *libaffa* library which is a C++ Affine Arithmetic library.

Affine example: Basic usage of Affine module

Interval example: Basic usage of Interval module

Conversion example: Conversion between IA and AA

Precision example 1: Use of precision module with *affapy*

Precision example 2: Use of precision module with decorators

1.4.1 Example 1

A simple example of using the lib

This example is a simply way to use the *affapy* library.

1.4.2 Example 2

Time performances between AA and IA model

This example shows the time performances between AA and IA. It permorms calculations on the following function:

$$x_1, x_2 \mapsto 1 + (x_1^2 - 2)x_2 + x_1 x_2^2$$

The result of this example is that AA is slower than IA.

Usage:

```
python3 example5.py [lbound1] [ubound1] [lbound2] [ubound2] [boxn]
```

- lbound1: lower bound of the interval 1 (default: 10)
- ubound1: upper bound of the interval 1 (default: 100)
- lbound2: lower bound of the interval 2 (default: 10)
- ubound2: upper bound of the interval 2 (default: 100)
- boxn: number of boxes (default: 1000)

1.4.3 Example 3

Underflow demonstration

This example shows that with a little uncertainty, the model of AA can return a huge interval after calculations.

1.4.4 Example 4

Comparison AA vs IA

We consider the function:

$$\frac{sin(x)^2 cos(x) - 4}{\sqrt{x}}$$

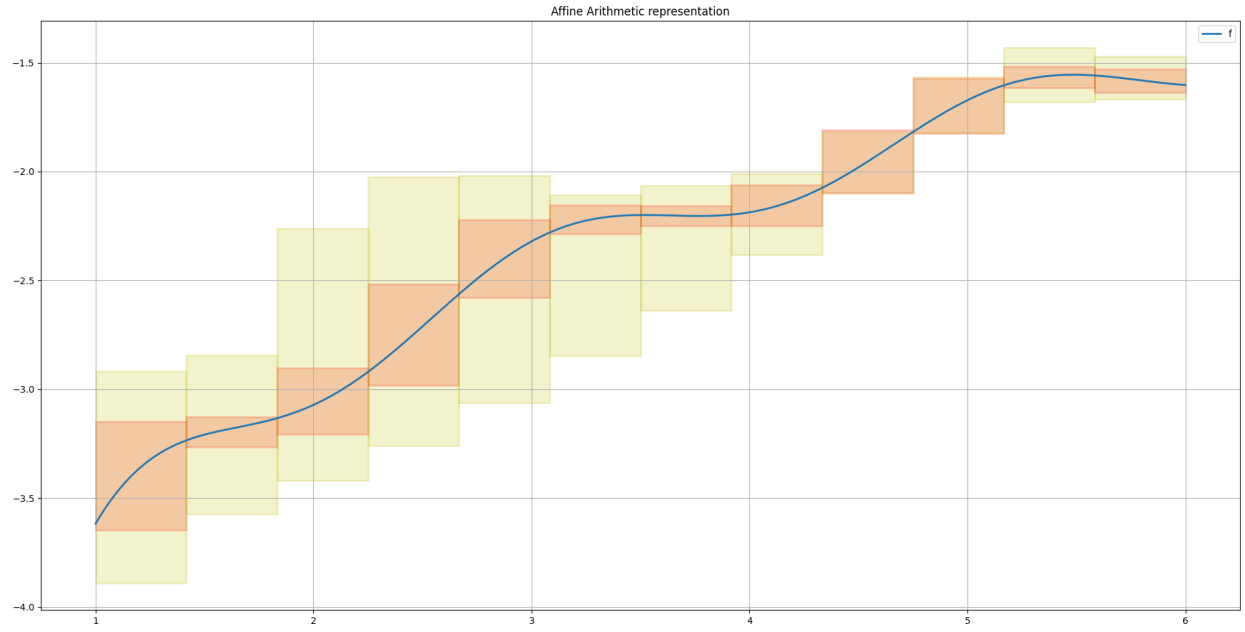
We consider the interval [lbound, ubound] (default: [1, 6]) and a subdivision of this interval containing n boxes (default: 12). The function is evaluated for each box with AA and IA. The result is plotted using matplotlib.

Usage:

```
python3 example4.py [lbound] [ubound] [boxn]
```

- lbound: lower bound of the interval (default: 1)
- ubound: upper bound of the interval (default: 6)
- boxn: number of boxes (default: 12)

You need to install *matplotlib* and *numpy* to run this example.



Output of example 4:

- in blue: the function
- in yellow: IA
- in orange: AA

1.4.5 Example 5

Comparison AA vs IA

We consider the function:

$$\frac{\sqrt{x^2 - x + \frac{1}{2}}}{\sqrt{x^2 + \frac{1}{2}}}$$

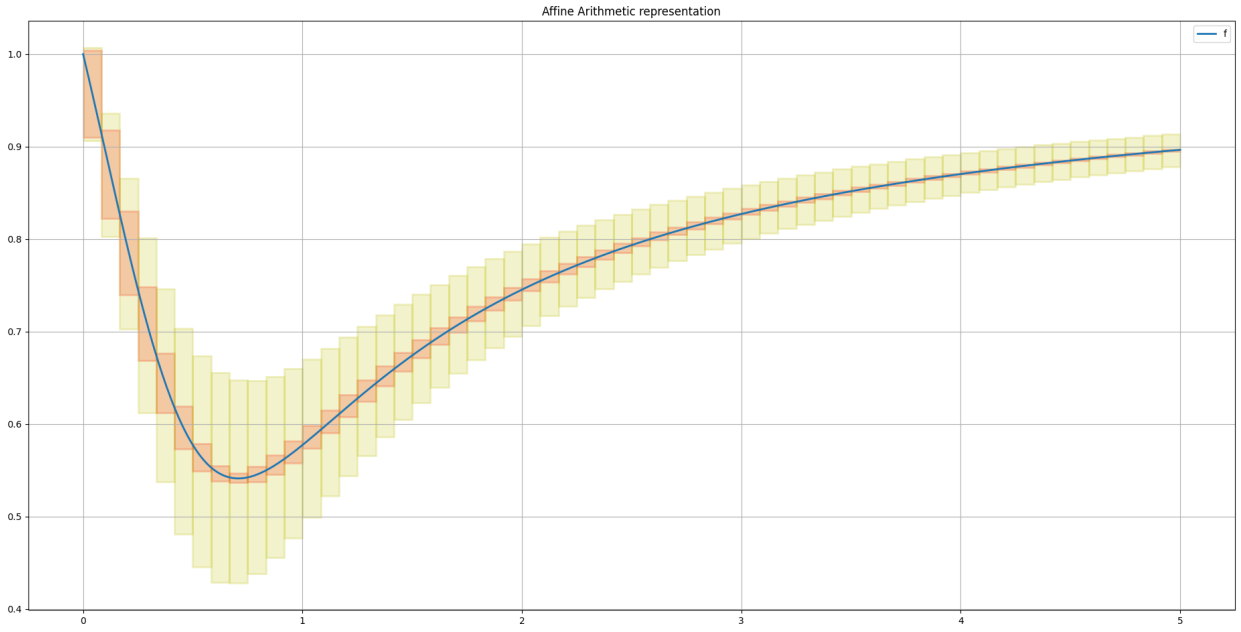
We consider the interval [lbound, ubound] (default: [0, 5]) and a subdivision of this interval containing n boxes (default: 60). The function is evaluated for each box with AA and IA. The result is plotted using matplotlib.

Usage:

```
python3 example5.py [lbound] [ubound] [boxn]
```

- lbound: lower bound of the interval (default: 0)
- ubound: upper bound of the interval (default: 5)
- boxn: number of boxes (default: 60)

You need to install *matplotlib* and *numpy* to run this example.



Output of example 5:

- in blue: the function
- in yellow: IA
- in orange: AA

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

C

D

E

I

L

M

O

P

R

S

T

W

X