# Interval arithmetic tutorial

## Setup

The `IntervalArithmetic.jl` package can be installed with

```
julia> using Pkg; Pkg.add("IntervalArithmetic")
```

Now you can import the package

```
using IntervalArithmetic
```

## Introduction

This tutorial will show you how to perform interval computations in Julia using the `IntervalArithmetic.jl` package. Interval arithmetic, opposed to traditional floating point arithmetic, uses intervals instead of single numbers. As floating point arithmetic introduces numerical error, interval arithmetic offers a framework to perform *rigorous* computations, meaning the output of the computation, be it solving a system of equations or an optimisation problem, will be an interval that is guaranteed to contain the correct solution. Before we dive into interval arithmetic, let's have a motivational example.

Observe the graph below, at first sight, it seems to have a small cuspid at $x = \frac{4}{3}$, zooming closer also seems to confirm it.

```
f(x) = (1/80) * log(abs(3*(1 - x) + 1)) + x^2 + 1 # hide
using Plots # hide
p1 = plot(0.5:0.01:2.0, f, leg=false) # hide
p2 = plot(1.2:0.0001:1.5, f, leg=false) # hide
plot(p1, p2, layout=(1,2), leg=false) # hide
```

However, the function in the figure is (this example is due to William Kahan, the father of floating point arithmetic)

$$f(x) = \frac{1}{80}\log\left(\left|3(1-x)+1\right|\right)+x^2+1$$

From the expression is now evident that the function has a vertical asympote at $x = \frac{4}{3}$ So what is going on? First, you may think that using more points might help, however the issue is nastier. First, floating point system cannot represent all real numbers, i.e. whenever you do real computations using floating points, you introduce a discretization error. Particularly, the number

$\frac{4}{3}$ itself is not exactly representable with floating point arithmetic, hence when you type `x=4/3` in your code, you actually get a number close to it. Observe

```
x = 4/3
f(x) = 1/80 * log(abs(3*(1 - x) + 1)) + x^2 + 1
@show f(x)
```

Hence using floating point arithmetic the function seems to be defined at `4/3`, which is obviously an absurd. Let us try to evaluate now the function at the previous and next floating point number.

```
f(prevfloat(x))
f(nextfloat(x))
```

Again, the function seems perfectly finite and well behaving. Conclusion: there is nothing you can do with floating point arithmetic to detect the singularity. Interval arithmetic will offer a framework to perform rigorous computations with real numbers and avoid this kind of issues. In this tutorial you will also see how interval arithmetic can detect the singularity for this function.

Interval arithmetic offers a methematical framework to performs *rigorous* computations. Computations with traditional floating point arithmetic introduce rounding error and hence the final result of the computation will be an approximate solution which is (hopefully!) close to the correct value. Interval arithmetic on the other side will output an interval which is *guaranteed* to contain the correct theoretical result of the computation. This tutorial will show you how to perform interval computations in Julia using the `IntervalArithmetic.jl` package. After this tutorial, you will be ready to harness the power of interval arithmetic and learn how to apply it to root finding, optimisation or differential equations.

# Defining intervals

An interval $I$ is a subset of $R$ in the form $I=\{x \in R \vee a \leq x \leq b\}$ and is denoted as $I=[a,b)$ and we say that $a$ and $b$ are the lower and upper bound. Note that the bounds must satisfy $a \leq b$. The IntervalArithmetic.jl package offers several ways to define an interval

## Dot notation

An interval $[a,b)$, with $a \leq b$ can be easily defined with the syntax `a..b`.

```
a = 1..2
@show a

b = 2..1
```

\note{If the upper bound is negative, then it should be within parentheses `-5..(-2)` or adding a space before it `-5.. -2`. The expression `-5..-2` will throw a syntax error.}

## Midpoint notation

An interval can be created also using the *midpoint notation*, i.e. $m \pm r$, where $m = \frac{a+b}{2}$ is the midpoint of the interval and $r = \frac{b-a}{2}$ is the radius. In julia, the symbol $\pm$ can be typed writing `\pm<TAB>`. This method is equivalent to `(m-r)..(m+r)`

```
2 ± 1
```

## `@interval` macro and `interval` method

Another way to define an interval is to use the `@interval(a, b)` macro or the `interval(a, b)` method. If called with a single input such as `@interval(a)` or `interval(a)`, they will create the degenerated interval $[a, a]$. Here some examples:

```
a = @interval(1,2)

b = interval(1, 2)

c = interval(1)

d = @interval 1

@show a, b, c, d
```

The macro is particularly useful when you want to create an interval from a more complicated expression, as demonstrated in the example below

```
@interval sin(0.2)+cos(1.3)-exp(0.4)
```

This is equivalent to `sin(interval(0.2))+cos(interval(1.3))-exp(interval(0.4))`.

There is a fundamental difference between `interval` and `@interval`: the former does not perform direct rounding on the boundaries. Suppose we want to create the interval $[0.1, 0.1]$. The number $0.1$ cannot be represented exactly with floating-point arithmetic, i.e. when the user types $0.1$ the computer automatically approximates to the closest representable number.

```
a = 0.1

@show big(a)
```

The following trick allows you to create a number as close as possible to the exact real number

```
correct = big"0.1"
@show correct
```

If we create the interval with the `@interval` macro or the `..`, the package will check whether the number is exactly representable in floating point arithmetic and if it is not, it will round down the lower bound and round up the upper bound, ensuring that $0.1$ is actually included in the interval. However, if the `interval()` method is used, no rounding is performed, resulting in the number not being contained in the interval.

```
I = interval(a)

II = @interval 0.1
III = a..a

@show correct ∈ I
@show correct ∈ II
@show correct ∈ III
```

Concluding, `..` and `@interval` are more sophisticated and ensure proper rounding if the number typed by the user is not exactly representable in floating point arithmetic. This introduces some computational overhead. `interval()` simply uses the floating point representation of the user input. This is faster, but the real numbers meant by the user might not necessarily be in the interval.

## Operations on intervals

In general, a binary operation $\circ$ between two intervals $X$ and $Y$ is defined as

$$X \circ Y = \left[ \{ x \circ y : x \in X, y \in Y \} \right)$$

where the brackets $\left[ A \right)$ denote the interval closure, i.e. the smallest interval containing the set $A$, for example if $A = \left[1,2\right) \cup \left[3,4\right)$ then $\left[A\right) = \left[1,4\right)$. Bearing this definition in mind, traditional arithmetic and set operations can be defined for intervals. These operations are already implemented in IntervalArihtmetic.jl and can be used with the traditional operators `+, -, *, /, ∩, ∪`. If you want to know how these are computed under the hood, check our interval functions grimoire! A few examples:

```
X = 1..2
Y = 3..4

@show X ∩ Y # typed \cap<TAB
@show X ∪ Y # TYPED \cup<TAB>
@show X + Y
@show X*Y
@show X^3
@show X/Y
```

Note that the operation $X/Y$ is finite only if $0 \notin Y$, otherwise the resulting interval will be unbounded

```
X = 1..2
```

```
Y = -1..1

Z = 0..2

@show X/Y

@show X/Z
```

It is good to notice that some standard properties of arithmetic operations do not hold in interval arithmetic, observe the following examples

```
X-X

X/X

Z*(X+Y)

Z*X+Z*Y
```

Hence in interval arithmetic $ X-X\neq0 $ and $ X/X\neq1 $. The last two operations tell us that $X\left(Y+Z\right)\subseteq X\,Y+X\,Z$, which is called *subdistributive properties*.

## Functions over intervals

Similarly to binary operations, the functions $f$ over an interval $X$ is defined as

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

Elementary functions have already been implemented in JuliaIntervals are are ready-to-go. If you are curious to see how they are computed under the hood, check our interval functions grimoire.

```
@show sin(0..2/3*π)

@show  log(-3.. -2)

@show log(-3..2)

@show  √(-3..4)
```

Now that we have discussed basic interval arithmetic operations, we are ready to return to Kahan's example. Let us create a small interval around `4/3`and evaluate the function there.

```
f(x) = (1/80) * log(abs(3*(1 - x) + 1)) + x^2 + 1
x = @interval 4/3
@show f(x)
```

As we can notice, interval arithmetic can successfully detect the singularity in the function, which we could not do with floating point arithmetic.

## A note on functions: overestimation

Suppose we want to evaluate the function $f(x)=x^2+3x-1$ over the interval $X=[-2,2)$.

```
f(x) = x^2+3x-1

X = -2..2

@show f(X)
```

According to this, $f(X)=[-7,9)$. However, it can be verified that the real range of the function over $X$ is $[-3.25,9) \subset [-7,9)$. In this case we *overestimate* the range of the function.

The reason for this overestimation is known as *dependence problem*. If in a function $f$ the same variable appears multiple times, then the interval evaluation of the function may lead to overestimation. A simple solution to overcome this problem would be to divide the interval in smaller disjoint intervals, evaluate the function on each interval and compute the union of the resulting intervals. For example, since $[-2,2)=[-2,-1.5) \cup [-1.5,2)$ we obtain

```
@show f(-2.. -1.5) ∪ f(-1.5.. 2)
```

As can be noticed, the overestimate is now reduced. Here you will find a more detailed discussion on how to estimate the range of a function and deal with function overestimates.

# Interval boxes

Interval boxes generalize the concept of interval to higher dimensions. More formally an interval box $X$ is a subset of $R^n$ defined as the cartesian product of $n$ intervals, i.e.

$$X = X_1 \times \cdots \times X_n$$

In julia, interval boxes can be created using the constructor `IntervalBox`, which takes the single intervals `X1,X2,...,Xn` as arguments. For the special case when $X_1=\cdots=X_n$ you can also use the shorter signature `IntervalBox(x, n)`. Observe the following examples

```
X1 = IntervalBox(1..2, 2..3, 3..4)
X2 = IntervalBox(1..2, 3..4)
X3 = IntervalBox(1..2, 5)

@show X1
@show X2
@show X3
```

Note that an interval box is a vector of intervals and as such, only operations which are well defined for vectors (addition and scalar multiplication) are implemented. Particularly, product of interval boxes or elementary functions are not well defined. If you want to apply a function to each component of the box, use broadcasting.

```
X1 = IntervalBox(1..2, 2..3)
X2 = IntervalBox(-1..1, 0..2)

@show X1 + X2
@show sin.(X1)
```

2D interval boxes are particularly handy for visualization. We can give the function `plot` (and `plot!`) a 2D interval box, or an array of boxes, to visualize these boxes in the cartesian plane. Recall our example from before

```
f(x) = x^2 + 3x - 1
X = -2..2
f1 = f(X)

X2 = [-2.. -1.5, -1.5..2]
f2 = f.(X2)
```

We had discussed that evaluated the function over the inverval overestimates the range, and splitting the interval helps reducing the overestimate. To help visualizing this, we first create the interval boxes $X \times f(X)$, note you can use broadcasting to create an array of boxes from arrays of interval

```
box1 = IntervalBox(X, f1)
box2 = IntervalBox.(X2, f2)
```

now we can plot the boxes to visualize the overestimate

```
using Plots
plot(box1)
plot!(box2)
plot!(f, -2, 2, leg=false, color=:black, linewidth=2)
```

As you can see, evaluating the range using two boxes reduces the overestimate of the range.

# Display modes

There are several useful output representations for intervals, some of which we have already touched on. The display is controlled globally by the `setformat` function, which has the following options, specified by keyword arguments:

- `format`: interval output format
  - `:standard`: output of the form `[1.09999, 1.30001]`, rounded to the current number of significant figures (default)
  - `:full`: output of the form `Interval(1.0999999999999999, 1.3)`, as in the `showfull` function
  - `:midpoint`: output in the midpoint-radius form, e.g. `1.2 ± 0.100001`
- `sigfigs`: number of significant figures to show in standard mode. Default value is 6

- `decorations` (boolean): whether to show decorations or not. Default false.

Here are some examples

```
a = sqrt(2) .. sqrt(3)
@show a

setformat(:full)
@show a

setformat(:midpoint)
@show a
setformat(:standard)

setformat(sigfigs=10)
@show a
```

# Advanced options: precision and rounding modes

## Changing bounds precision

By default, the methods described above use `Float64` for the lower and upper bound of the interval. You can verify this using the function `precision(Interval)`

```
precision(Interval)
```

The result is a tuple. The first value tells us that the lower and upper bounds of the interval are stored as 64-bits floating points. The second value is the precision of `BigFloat` used for interval computations.

You can change the precision of the bounds using `setprecision(Interval, T)`, where `T` can be a type (e.g. `Float64`) or an integer number indicating the number of bits.

```
setprecision(Interval, 256)
@show precision(Interval)
@interval π
```

The subscript `256` tells us that the bounds of the interval are `BigFloat` with 256 bits. We can set the precision back to default value with

```
setprecision(Interval, Float64)
```

## Rounding modes

As we discussed before, interval arithmetic computations use direct rounding. This means that when computing `f(X)`, where `X` is an interval, the lower bound is rounded down and the upper bound is rounded up. This is achieved using th \elink{CRlibm} {https://github.com/JuliaIntervals/CRlibm.jl} library. While this ensures the inverval is as tight as

possible, it also introduces some computational overhead. An alternative rounding method is to perform calculations using the (standard) RoundNearest rounding mode, and then widen the result by one machine epsilon in each direction using prevfloat and nextfloat. This produces a wider interval, but significantly speeds up computations. The rounding mode can be changed using `setrounding(Interval, mode)` function. The `mode` parameter can have values

- `:fast` tight (correct) rounding with errorfree arithmetic via FastRounding.jl. Faster than other tight methods, but might be incorrect for extreme inputs
- `:tight` tight (correct) rounding with improved errorfree arithmetic via RoundingEmulator.jl
- `:accurate` fast "accurate" rounding using prevfloat and nextfloat, slightly wider than needed but faster
- `:slow` tight (correct) rounding by changing rounding mode.
- `:none` no rounding (for speed comparisons; no enclosure is guaranteed) For example

```
setrounding(Interval, :accurate)
```

will use round to nearest during computations and then use `prevfloat` and `nextfloat`for the interval bounds. You can return to the default mode as follows

```
setrounding(Interval, :slow)
```

You can check the rounding mode you are currently using with the `rounding`function

```
rounding(Interval)
```

---

*This notebook was generated using Literate.jl.*