The Basics

In this notebook we go over the very basics of interval arithmetic and computer assisted proofs.

```
using Arblib , IntervalArithmetic , Plots , PlutoUI , SpecialFunctions
```

64

```
setprecision(Arb, 64)
```

In this tutorial we will be using a mix of the two packages `IntervalArithmetic.jl` and `Arblib.jl`. The first is a Julia implementation of interval arithmetic and the latter is a Julia wrapper round the **Arb** library.

They both come with their pros and cons, `IntervalArithmetic.jl` is probably easier to use to start with and can be much faster in some cases. `Arblib.jl` has a lot of functionality not offered by `IntervalArithmetic.jl`, in particular it supports a lot more special functions and also the computation of series expansions.

Going more in detail about the differences between these two packages is outside the scope of this tutorial, but feel free to ask me about it.

Enclosing a function

The most basic thing interval arithmetic allows us to do is to enclose the range of a function. Instead of evaluating a function at a point and getting an approximation of the result we'll evaluate a function over an interval and as result get an interval certain to contain the range of the function on the first interval.

We can do this at individual point

```
(3.14159, 1.22465e-16)
```

```
Float64(π), sin(π)
```

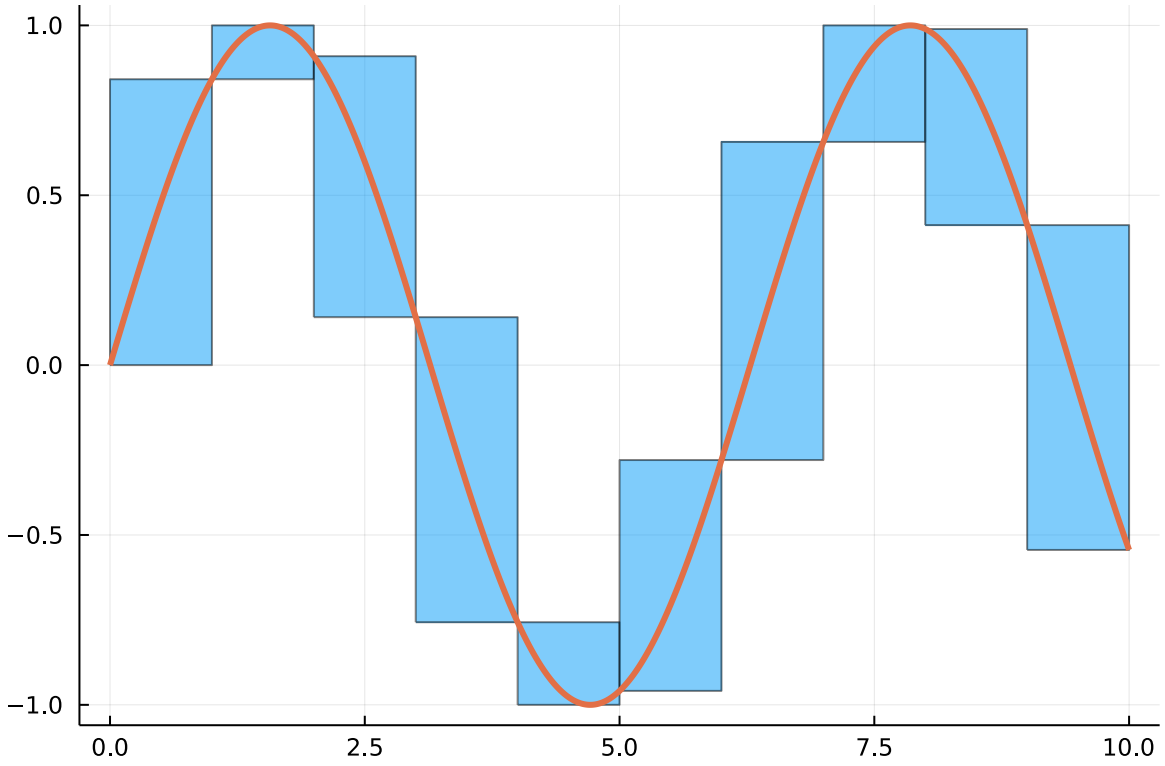
```
([3.14159, 3.1416], [-3.21625e-16, 1.22465e-16])
```

```
Interval(π), sin(interval(π))
```

But it is maybe easier if we visualise it by consider a whole range of intervals

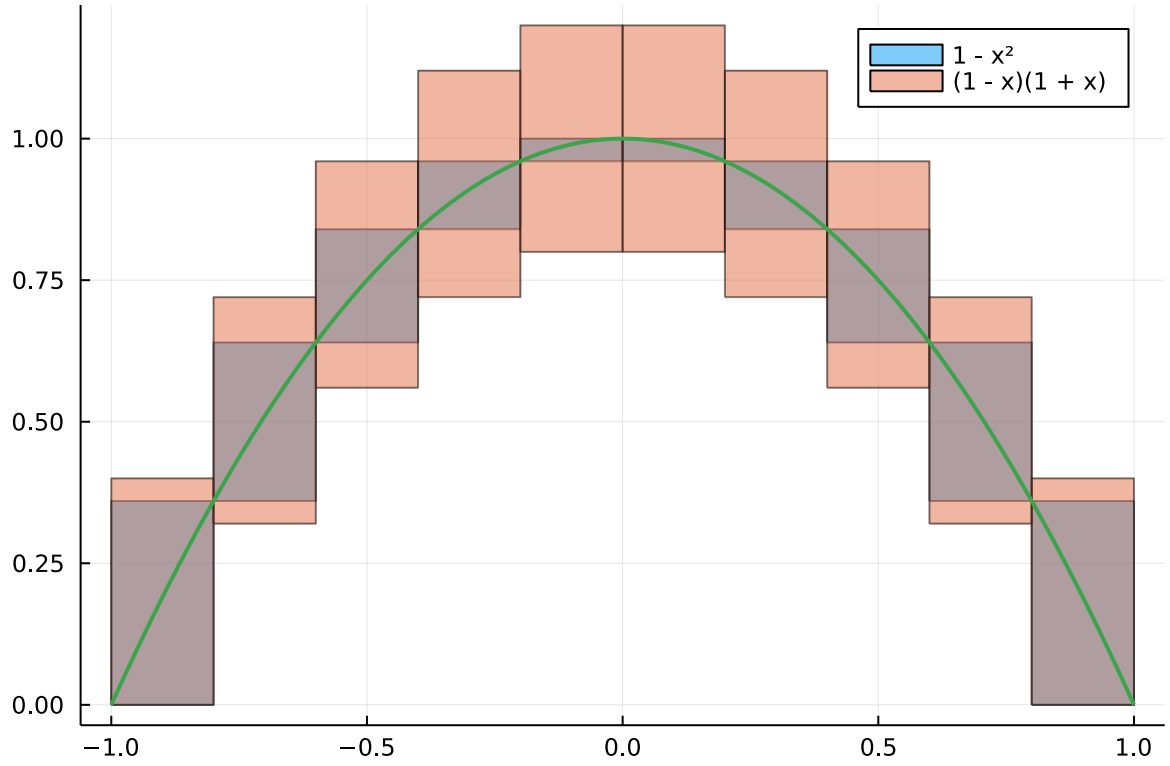
10

```
@bind num_intervals Slider(1:100, default = 10, show_value = true)
```




```
let n = num_intervals
    f = sin
    a = 0
    b = 10
    intervals = IntervalArithmetic.mince(Interval(a, b), n)
    enclosures = f.(intervals)
    pl = plot(IntervalBox.(intervals, enclosures), legend = :none)
    plot!(pl, range(a, b, length = 1000), f, linewidth = 3, label = "")
end
```

This makes it easy to see how different forms of the function affects the resulting enclosure.




```
let n = num_intervals
    f = x -> 1 - x^2
    g = x -> (1 - x) * (1 + x)
    a, b = -1, 1
    intervals = IntervalArithmetic.mince(Interval(a, b), n)
    pl = plot(IntervalBox.(intervals, f.(intervals)), label = "1 - x²")
    plot!(pl, IntervalBox.(intervals, g.(intervals)), label = "(1 - x)(1 + x)")
    plot!(pl, range(a, b, length = 1000), f, linewidth = 2, label = "")
end
```

Isolating roots

Once commong thing that you have to do when contstructing a computer assisted proof is to isolate roots of functions. If you have a function $f : \mathbb{R} \to \mathbb{R}$ then this is very straigt forward.

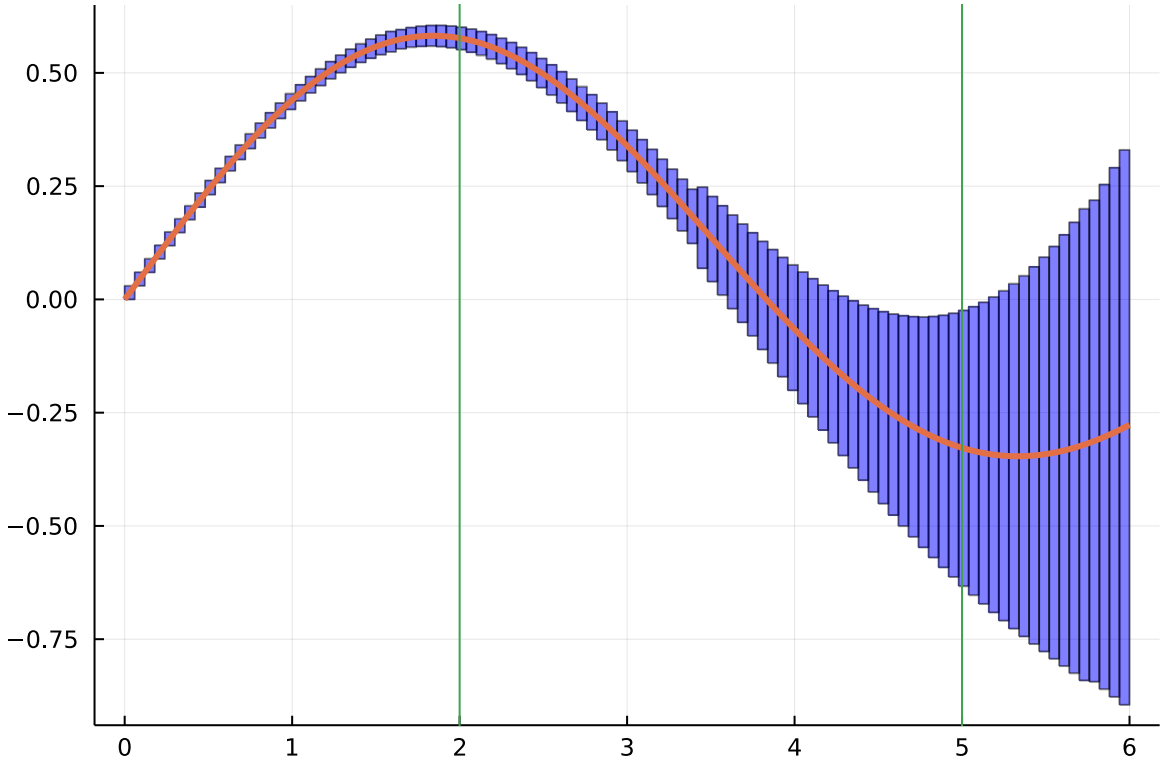
We will prove that the following function has a unique root on the interval [2, 5]. Since this function is not implemented by `IntervalArithmetic.jl` we will use `Arblib.jl` instead.

```
f = besselj1 (generic function with 7 methods)
```

```
f = besselj1
```

```
(2.000000000000000000, 5.000000000000000000)
```

```
lower, upper = Arb(2), Arb(5)
```




```
let n = 100
    a, b = Arb(0), Arb(6)
    intervals = mince(Arb((a, b)), n)
    enclosures = f.(intervals)
    pl = boxplot(intervals, enclosures)
    plot!(pl, range(a, b, length = 1000), f, linewidth = 3, legend = :none)
    vline!(pl, [lower, upper])
end
```

The first step is to prove that it has **at least** one root in the interval. To do that we just check that the sign of `f` is different on the left and the right endpoint. By the mean value theorem it would then of course follow that it has at least one root, though it doesn't exclude the possibility of multiple roots.

```
f_lower = [0.5767248077568733872 +/- 9.27e-20]
```

```
f_lower = f(lower)
```

```
f_upper = [-0.327579137591465222 +/- 2.01e-19]
```

```
f_upper = f(upper)
```

```
true
```

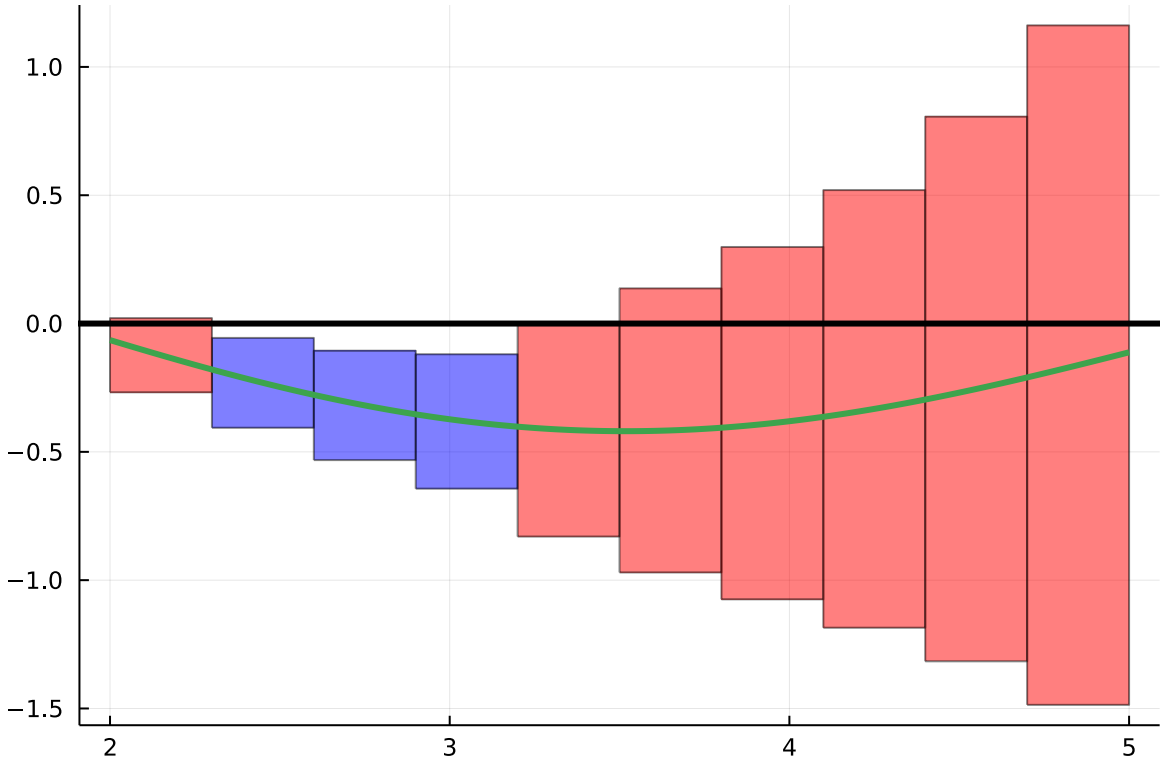
```
Arblib.ispositive(f_lower) && Arblib.isnegative(f_upper)
```

To prove that there is a unique root in the interval we will prove that the function is monotone on the whole interval. For that we need the derivative of the function. While the derivative can be computed automatically by the computer (using Taylor series) we will here give it explictly instead.

```
df = #5 (generic function with 1 method)
```

```
df = x -> (besselj0(x) - besselj(oftype(x, 2), x)) / 2
```

10




```
let n = n_monotone
    intervals = mince(Arb((lower, upper)), n)
    enclosures = df.(intervals)

    not_zero = .!Arblib.contains_zero.(enclosures)

    pl = boxplot(intervals[not_zero], enclosures[not_zero], color = :blue)
    boxplot!(pl, intervals[.!not_zero], enclosures[.!not_zero], color = :red)
    plot!(pl, range(lower, upper, length = 1000), df, linewidth = 3, legend = :none)
    hline!([0], linewidth = 3, color = :black)
end
```

So now we have managed to prove that there is at least one zero and that the function is monotone, we can conclude that there is a unique zero in the interval!

Integration

The final part of this first introduction is about integration. There are many different methods for rigorous integration. We will start with the most naive zeroth order method that you could imagine. The performance is, not suprisingly, horrible.

Then we will briefly showcase the integrator in Arb, which is convinient to use from `Arblib.jl`.

Consider the following function which we want to integrate from `0` to `5`.

- Using a box plot
- Using higher order methods
- 10.1007/978-3-319-96418-8_30

```
g = sin (generic function with 26 methods)
```

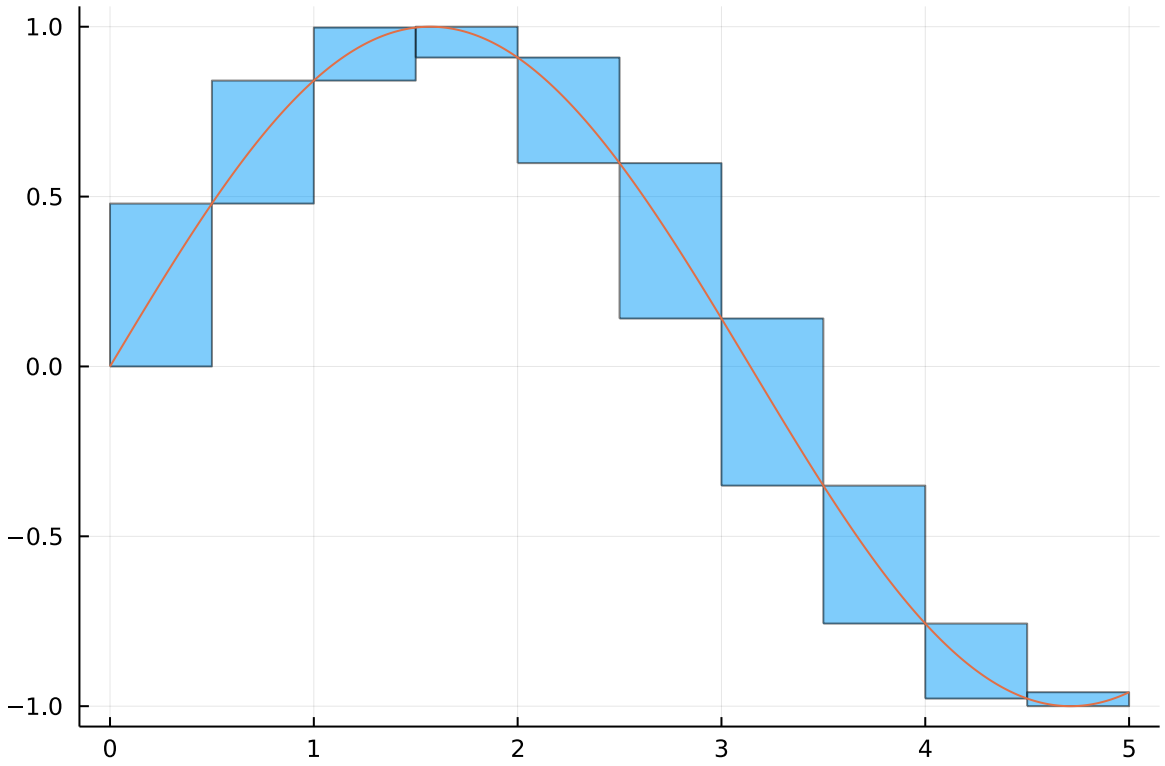
```
g = sin
```

```
(0, 5)
```

```
a, b = 0, 5
```

10

```
@bind n_integration Slider(10:10:10000, default = 10, show_value = true)
```



An extremely naive enclosure of the integral is given by bounding the integral on each interval by just using the enclosure of the function on the interval.

```
integral_naive = [-0.0576649, 1.45039]
```

```
integral_naive = let n = n_integration
    intervals = IntervalArithmetic.mince(Interval(a, b), n)
    sum(diam(interval) * g(interval) for interval in intervals)
end
```

There are many higher order methods that are simple to implement that are much better to use. We will use the integration routine that is implemented in Arb. It is based on Petras algorithm. It doesn't need to compute any derivatives of the function but instead uses analytic properties of the function. The paper for it can be found on **arXiv**.

```
[0.71633781453677374 +/- 6.05e-18]
```

```
Arblib.integrate(g, a, b)
```

Utility methods

Here we define some utility methods that are used in the code above.

```
mince (generic function with 1 method)
```

```
function mince(x::Arb, n::Integer)
    a, b = getinterval(x)
    intervals = [zero(a) for _ = 1:n+1]
    Arblib.set!(intervals[1], a)
    Arblib.set!(intervals[n+1], b)
    diff = (b - a) / n
    for i = 2:n
        Arblib.add!(intervals[i], intervals[i-1], diff)
    end

    balls = Vector{Arb}(undef, n)
    for i = 1:n
        balls[i] = Arb((intervals[i], intervals[i + 1]))
    end

    return balls
end
```

```
boxplot (generic function with 1 method)
```

```
function boxplot(xs, ys; color = :blue)
    xs = [Interval(getinterval(BigFloat, x)...) for x in xs]
    ys = [Interval(getinterval(BigFloat, y)...) for y in ys]
    plot(IntervalBox.(xs, ys); color)
end
```

```
boxplot! (generic function with 1 method)
```

```
function boxplot!(pl, xs, ys; color = :blue)
    xs = [Interval(getinterval(BigFloat, x)...) for x in xs]
    ys = [Interval(getinterval(BigFloat, y)...) for y in ys]
    plot!(pl, IntervalBox.(xs, ys); color)
end
```