Interval root finding tutorial

Setup

The `IntervalRootFinding.jl` package can be installed with

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

Once the package is installed, it can be imported. Note that you will need also the `IntervalArithmetic.jl` package.

```
using IntervalArithmetic, IntervalRootFinding
```

Introduction

The `IntervalRootFinding.jl` can be used to rigorously compute the zeros of a function $f: R^m \to R^n$ over a given interval (or interval box) $X \subset R^m$.

Basic usage

To get started, let's compute the roots of the simple function $f(x) = x^2 - 2x$ over the interval $[-4, -4]$. To fullfil the task, we can use the function `roots` from the `IntervalRootFinding` package which has syntax

`rts = roots(f, X, method)`, \where

- `f` is the function whose roots we want to find
- `X` is the interval over which we look for the roots
- `method` is a third optional parameter that specifies what method is used to compute the roots. The following methods are supported: Newton (default), Krawczyk and Bisection.

```
f(x) = x^2-2x
X = -4..4

rts = roots(f, X) # using newton method
```

As can be noticed, the result of the function is an array of elements of type `Root`. Each Root-element has two fields: `interval`, containing the interval contain a zero of the function, and `status`, a "label" telling whether the interval is guaranteed to contain a single root (`:unique`) or whether the interval may possibly contain multiple or zero roots (`:unknown`). It is good to notice that only Newton and Krawczyk methods can make some considerations on the uniqueness of the roots. If Bisection method is used, the status of all roots will automatically be unknown, as the following example shows.

```
roots(f, X, Bisection)
```

Note also how the Bisection method may return some extra intervals not containing the roots.

Explicit derivatives

Both Newton and Krawczyk method need to compute the derivative of the function. This is done under the hood using the `ForwardDiff.jl` package. You can help the solver by explicitly giving the derivative of the function as a second parameter. This is particularly useful if ForwardDiff is not able to compute the derivative.

```
roots(log, x->1/x, -2..2)
```

Note that this will (generally) speed up the computations a little, but it will not affect the accuracy of the result

```
using BenchmarkTools
@btime roots(log, x->1/x, -2..2)

@btime roots(log, -2..2)
```

Multidimensional example

The function `roots` works also for multidimensional functions $R^m \to R^n$, with a few adjustments:

- the function should return a `SVector`, an efficient data structure for arrays whose size is known at compile time. SVector requires the package `StaticArrays`.
- the function can take as input an array or a tuple of scalars
- The roots are now searched over an `IntervalBox`

```
using StaticArrays
g( (x, y) ) = SVector(sin(x), cos(y))
X = IntervalBox(-3..3, 2)

rts = roots(g, X)
```

Similarly to 1D, if Newton or Krawczyk method are used, the solver can be helped by explicitly providing the Jacobian, which should be a `SMatrix`

```
function dg( (x, y) )
    return SMatrix{2, 2}(cos(x), 0, 0, -sin(y))
end

@btime roots(g, dg, X)
@btime roots(g, X)
nothing # hide
```

Tolerance

By default, the roots are found with a tolerance of `1e-15`. This can be changed by giving the solver an extra parameter. A Higher tolerance can significantly speed up the computations. However, as a drawback, more intervals might have an `:unknown` status. Observe the following example

```julia
h(x) = cos(x) * sin(1 / x)

@btime roots(h, 0.05..1)

@btime roots(h, 0.05..1, Newton, 1e-2)

@btime roots(h, 0.05..1, Newton, 1e-1)
```

This notebook was generated using Literate.jl.