Activity 1 - Babylonian square root (45 minutes)

This activity is inspired by Alan Edelman's "Automatic Differentiation in 10 minutes with Julia" talk, and Carsten Bauer's "Julia for HPC Course @ UCL ARC" workshop.

Heron's algorithm

In ancient Greece, the mathematician Hero of Alexandria described what is now known as Heron's method or the Babylonian method of calculating the square root of a number, $\sqrt{x}$.

For calculating the square root of a positive number x, Heron's method consisted in iteratively computing

$$y_n = \frac{1}{2}\left(y_n + \frac{x}{y_n}\right),$$

until a desired accuracy is achieved, with $\lim_{n\to\infty} y_n = \sqrt{x}$ and $y_0 > 0$. This method is very good, and converges quadratically, as it is just Newton's root finding method for $f(y) = y^2 - x$.

An implementation of this algorithm in Julia can be seen below:

```julia
function sqrt_babylonian(x, N = 10)
    y = (1 + x) / 2 # First iteration, y0 = 1
    for i = 2:N
        y = (y + x / y) / 2
    end
    return y
end
```

Numerical precision

Does this algorithm work?

> **Task 1:** Confirm that the iterative algorithm converges. For this, compare the output of `sqrt(big"2.0")` and `sqrt_babylonian(2.0, N)` for `N` increasing from `1` to `10`.

In the definition of `sqrt_babylonian`, we never forced `x` to be of any specific type. This is called a generic implementation.

> **Task 2:** Do the same as in task 1, but vary the data type of the input. Specifically, use `Float16(2.0)`, `Float32(2.0)`, `Float64(2.0)` and `sqrt(big"2.0")`.

The combination of **generic code** and **special data types** can lead to "emergent" features. Below, we'll consider three simple but hopefully somewhat exciting examples that will make our `sqrt_babylonian`

- compute not only the square root itself but also its **derivative**,
- produce an **analytical expression** that approximates the square root, and
- **propagate uncertainty** in the input (according to linear error propagation theory) to the output.

And all of this **without modifying our implementation**.

Automatic differentiation (AD)

A powerful number type invented by Clifford in 1873 is the **dual number**. One application of these numbers is known as **forward-mode automatic differentiation (AD)**.

Dual numbers are expressions of the form $a_x + b_\epsilon \varepsilon$, where the symbol ε satisfy $\epsilon^2 = 0$.

```
struct D <: Number # Dual number
    x::Float64 # Value
    ϵ::Float64 # Derivative
end
```

Of course, this numbers satisfy the **addition rule**

$$(a_x + a_\epsilon \varepsilon) \pm (b_x + b_\epsilon \varepsilon) = (a_x \pm b_x) + (a_\epsilon \pm b_\epsilon)\varepsilon,$$

that can easily be translated to Julia code:

```
# Extending + and - symbols from Base
Base.:+(a::D, b::D) = D(a.x + b.x, a.ϵ + b.ϵ)
Base.:-(a::D, b::D) = D(a.x - b.x, a.ϵ - b.ϵ)
```

The use of `Base.:+` is to extend the symbol `+` from Base, this notation explicitly shows from which module the function is being extended. An alternative notation would be to import the symbol to be extended with `import Base: +` and then `+(x::D, y::D) = ....`

As $\epsilon^2 = 0$, dual numbers satisfy the **product rule**

$$\begin{aligned}(a_x + a_\epsilon \varepsilon)(b_x + b_\epsilon \varepsilon) &= a_x b_x + a_x b_\epsilon \varepsilon + a_\epsilon b_x \varepsilon + a_\epsilon b_\epsilon \epsilon^2 \\ &= (a_x b_x) + (a_\epsilon b_x + a_x b_\epsilon)\varepsilon.\end{aligned}$$

Look at the term multiplied by ε. Does it look similar to the derivative product rule $(fg)' = f'g + fg'$?

```
Base.:*(a::D, b::D) = D(a.x * b.x, a.x * b.ϵ + a.ϵ * b.x)
```

The **quotient rules** is derived by rationalizing the expression

$$\begin{aligned}\frac{a_x + a_\epsilon \varepsilon}{b_\epsilon + b_\epsilon \varepsilon} &= \frac{(a_x + a_\epsilon \varepsilon)(b_x - b_\epsilon \varepsilon)}{(b_x + b_\epsilon \varepsilon)(b_x - b_\epsilon \varepsilon)} \\ &= \frac{a_x}{b_x} + \frac{a_\epsilon b_x - a_x b_\epsilon}{b_x^2}\varepsilon.\end{aligned}$$

Look at the term multiplied by ε. Does it look similar to the derivative quotient rule $(f/g)' = (f'g - fg')/f^2$?

```
Base.:/(a::D, b::D) = D(a.x / b.x, (b.x * a.ϵ - a.x * b.ϵ) / b.x^2)
```

Additionally, we need to define how to convert and promote regular numbers to dual numbers

```
# Operations between Number and D <: Number convert Number to D
Base.promote_rule(::Type{D}, ::Type{<:Number}) = D
# How to convert Number (1 arg) to D (two args)
Base.convert(::Type{D}, x::Real) = D(x, zero(x)) # Real to Dual
```

Finally, to take the derivative of a function `f`

```
derivative(f::Function, x::Number) = f(D(x, one(x))).ϵ
```

✎ **Task 3:** What is the analytical derivative of $\sqrt{x}$? Remember your calculus class 😉.

Task 4: Check that `derivative(sqrt_babylonian, x)` indeed automagically gives the correct value of the derivative.
Feel free to try other functions/algorithms as well! Maybe something recursive like `pow(x, n) = n <= 0 ? 1 : x * pow(x, n-1)`?

Symbolics

Task 5: Verify that

$$\text{sqrt\_babylonian}(x, 3) = \frac{7x + 17x^2 + 7x^3}{8\,(1+x)\left(\frac{1}{4} + \frac{3}{2}x + \frac{1}{4}x^2\right)}.$$

Use the `Symbolics` package, in particular `@variables x` and `simplify`.

Uncertainty propagation

In the experimental sciences, numerical values (e.g., from measurements) are often subject to uncertainties due to systematic precision errors of the measurement devices. The Julia package Measurements.jl provides a number type and corresponding arithmetical operations that address this situation. Specifically, the package implements linear error propagation theory, which states that given a function $f(x)$ and an input value x_0 with uncertainty Δx_0, the uncertainty of $f(x_0)$ is given by

$$\Delta(f(x_0)) = \frac{\mathrm{d}f}{\mathrm{d}x}(x_0)\Delta x_0,$$

that is, the derivative of f evaluated at x_0 multiplied by the input uncertainty Δx_0.

Task 6: Try to run our sqrt algorithm with a `Measurement` as input, i.e. `sqrt_babylonian(2.0 ± 0.1)`. Does it work? What uncertainy do you get for the result?
Hint: you can get the `±` by typing `\pm` and then pressing the “TAB” key.

Task 7: It never hurts to check for correctness: Does the obtained uncertainty match the formula above for $f = \text{sqrt}$?