Activate supermagical presentation mode

Crash Course in VMC

30 minutes of Automatic Differentiation

(in Julia)

Filippo Vicentini

filippo.vicentini@epfl.ch

Setup

This is a fancy, super-interactive notebook on Automatic Differentiation.

No netket here... :😭

If you are curious... How to run this notebook:

- Install Julia: see **https://julialang.org/downloads/**
  - MacOs: 😊 `brew cask install julia` 😊
  - Linux: 😱 Check https://julialang.org/downloads/platform/#linux*and*freebsd
    - don't use `apt intstall julia` unless you have recent distro
  - Don't tell Giuseppe
- Install dependencies (only the first time)

```
julia --project=. -e "using Pkg; Pkg.instantiate()"
```

- Run the notebook

```
julia --project=. -e "using Pluto; Pluto.run()"
```

```julia
begin
    using Plots
    using PlutoUI
    using ForwardDiff
end
```

Optimising a Cost function

A cost function is a *Scalar* *real valued* function

$$\mathcal{C} : \mathbb{R}^N \rightarrow \mathbb{R}$$

And to optimise it we usually need to compute it's gradient

$$\vec\nabla\mathcal{C} : \mathbb{R}^N \rightarrow \mathbb{R}^N$$

$$\mathcal{W} \rightarrow \vec\nabla\mathcal{C}(\mathcal{W})$$

```
md"## Optimising a Cost function

A cost function is a _Scalar_ _real valued_ function

$\mathcal{C} : \mathbb{R}^N \rightarrow \mathbb{R}$

And to optimise it we usually need to compute it's gradient

$\vec\nabla\mathcal{C} : \mathbb{R}^N \rightarrow \mathbb{R}^N$
$\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \mathcal{W} \rightarrow \vec\nabla\mathcal{C}(\mathcal{W})$
"
```

Gradients: the general case

Let's now consider a slightly more general case

$$f : \mathbb{R}^{D_0} \rightarrow \mathbb{R}^{D_1}$$

The first-order differential structure in the neighborhood of the point $x \in \mathbb{R}^{D_0}$ is encoded into the *Jacobian Matrix*

$$(\mathcal{J}_f)^i_j(x) = \frac{(\partial f)^i}{\partial x^j}(x)$$

Given another function

$$g : \mathbb{R}^{D_1} \rightarrow \mathbb{R}^{D_2}$$

the Jacobian matrix for the composed function $h = g \circ f : \mathbb{R}^{D_0} \rightarrow \mathbb{R}^{D_2}$ can be written as:

$$(\mathcal{J}_h)^i_j(x) = \frac{(\partial g)^i}{\partial y^k}(f(x))\frac{(\partial f)^k}{\partial x^j}(x)$$

$$(\mathcal{J}_h)^i_j(x) = (\mathcal{J}_g)^i_k(f(x))(\mathcal{J}_f)^k_j(x)$$

This is the chain rule

Our objective: derivatives

In the following: How to compute the derivative of a function $f(x) : \mathbb{R} \rightarrow \mathbb{R}$ on a computer.

The derivative is defined as:

$$f'(x) = \frac{df}{dx}(x) = \lim_{\delta\rightarrow 0}\frac{f(x+\delta) - f(x)}{\delta}$$

```
md"""# Our objective: derivatives
In the following: How to compute the derivative of a function $f(x) : \mathbb{R} \rightarrow \mathbb{R}$ on a computer.

The derivative is defined as:

$f'(x) = \frac{df}{dx}(x) = \lim_{\delta\rightarrow 0}\frac{f(x+\delta) - f(x)}{\delta}$
"""
```

Roundoff errors

The easiest way to compute the derivative is by taking a small δ and using the formula above.

Choosing a good δ is hard:

- if δ too big $\rightarrow$ wrong result because the formula is asymptotic
- if δ too small $\rightarrow$ wrong result because of roundoff errors:

If I take a random (flating-point) number around 10^{-10}...

```
δ = 1.6244984317999655e-11
```

The *machine epsilon* of δ is the smallest number we can add to δ and obtain a number different from δ

```
3.2311742677852644e-27
```

```
eps(δ)
```

Usually with double precision (8 byte) floating point number we have 16 digits of precision ($25 - 10 \approx 16$) in the *low* digits.

```
δ₂ = 1.000000000016245
```

```
δ₂ = 1 + δ
```

You see that now δ_2 has lost information about the lowest digits of δ, because it's accurate only up to the 16-th digit.

```
2.220446049250313e-16
```

```
eps(δ₂)
```

```
δ̃ = 1.6245005340920216e-11
```

```
δ̃ = δ₂ - 1
```

Finite Differencing

So if i consider the simple function:

```
f(x) = exp(x) + sin(x);
```

and it's analytical derivative:

```
dfdx(x) = exp(x) + cos(x);
```

```
forward_difference(f, x, δ) = (f(x+δ) - f(x))/δ;
```

```
δ_arr =  Float64[1.0e-16, 1.45083e-16, 2.1049e-16, 3.05386e-16, 4.43062e-16, 6.
```

```
# I first compute the exact values using Arbitrary precision arithmetics
δ_arr = 10 .^ (range(-16, 0, length=100))
```

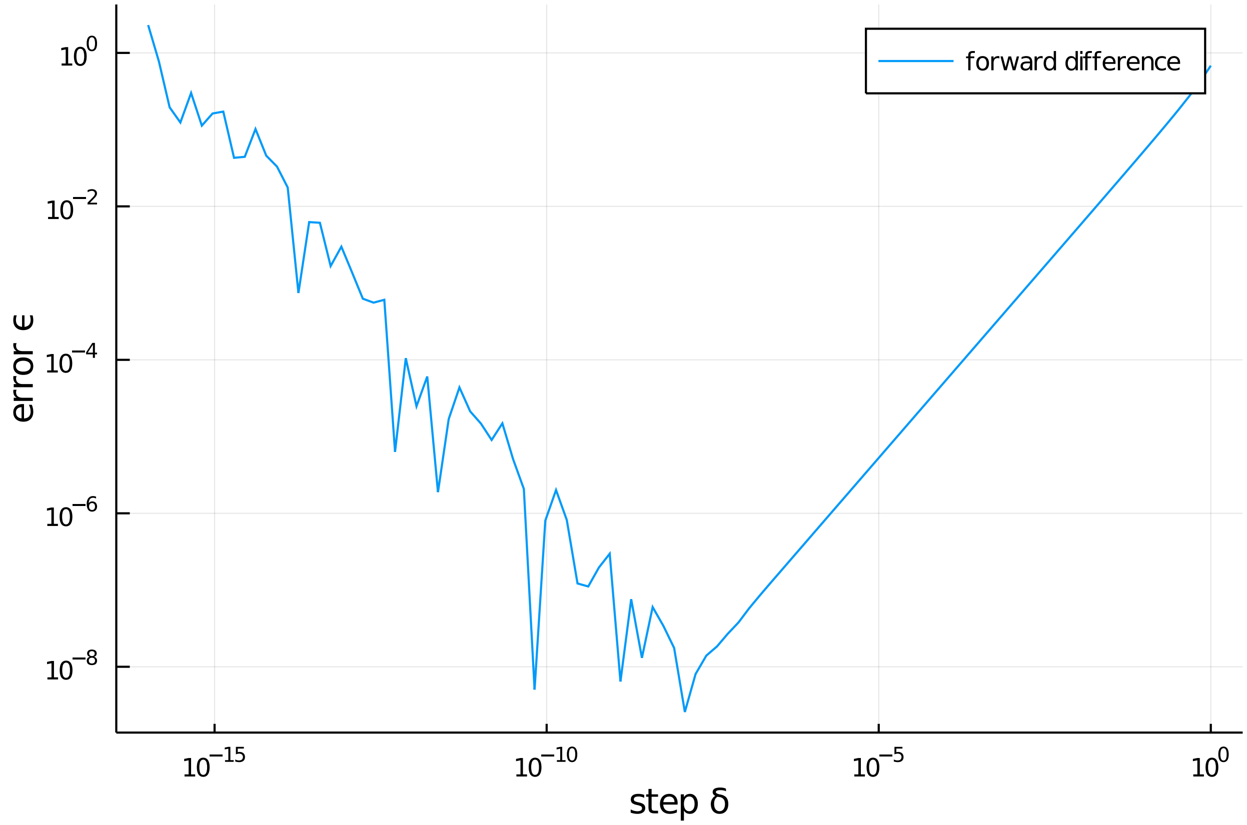
I select a point x_0 in it's domain and compute the derivative there: $...

x_0 =0.3

```
ϵ_fwd =  Float64[2.3052,  0.774728,  0.195412,  0.123907,  0.300559,  0.112811,  0.16
```

```
ϵ_fwd = diff_error(forward_difference, x₀, δ_arr)
```

```
pl =
```



```
pl = plot(δ_arr, ϵ_fwd, xscale=:log10, yscale=:log10, label="forward
difference",  xlabel="step δ", ylabel="error ϵ")
```

Finite Differencing: take 2

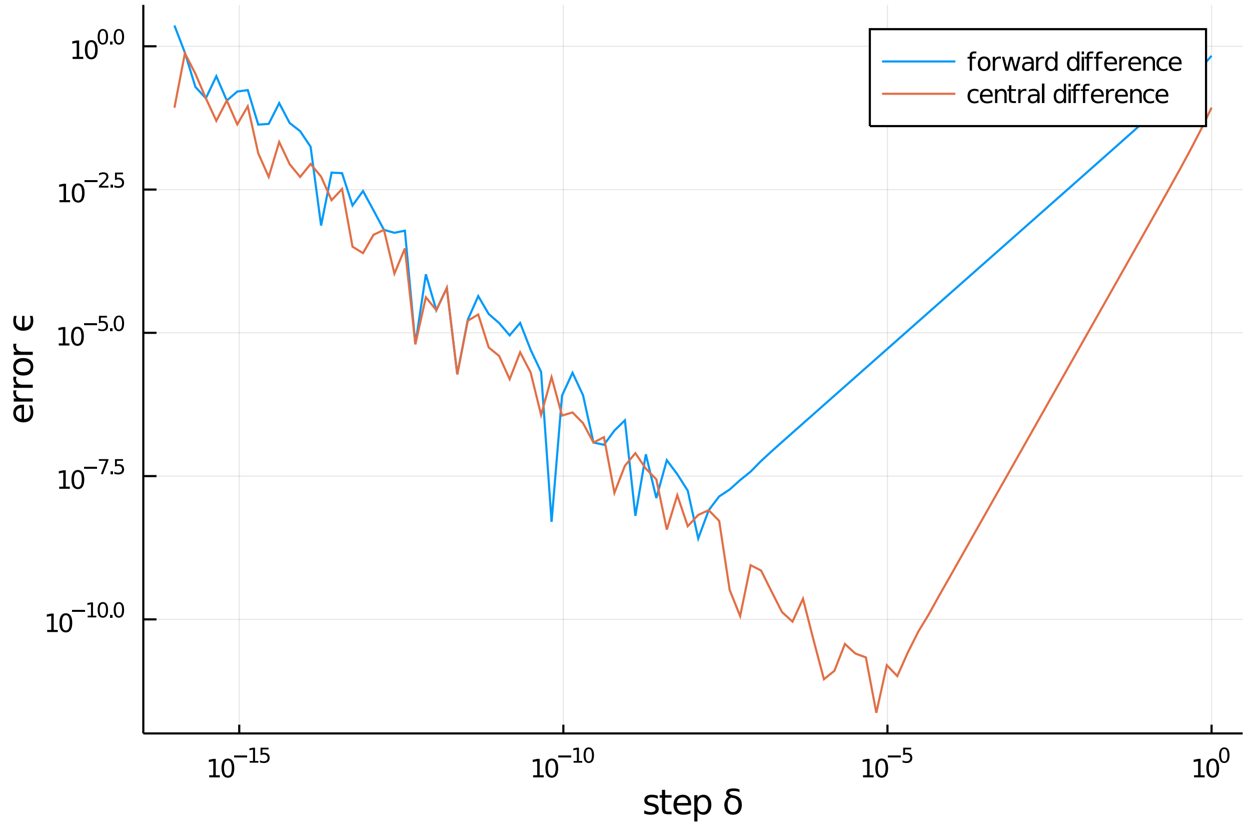
Instead of using the forward differencing formula, we can use more accurate central difference:

```
central_difference(f, x, δ) = (f(x+δ) - f(x-δ))/2δ;
```

```
ϵ_cnt =  Float64[0.0847492, 0.755739, 0.332034, 0.123907, 0.0499796, 0.112811,
```

```
ϵ_cnt = diff_error(central_difference, x₀, δ_arr)
```

```
pl2 =
```



```
pl2 = plot!(deepcopy(pl), δ_arr, ϵ_cnt, xscale=:log10, yscale=:log10,
label="central difference")
```

And this slope also depends on the function that we are evaluating...

Algebraic Approach: Complex step

The problem with finite differencing is that we are mixing our really small number with the really large number, and so when we do the subtract we lose accuracy.

- We want to keep the perturbation (f'(x)) and the value (f(x)) completely separate

$$f(x + i\delta) = f(x) + f'(x)i\delta + \mathcal{O}(\delta^2)$$

$$if'(x) = \frac{f(x+i\delta)-f(x)}{\delta} + \mathcal{O}(\delta)$$

If x is real and f is real-valued then if' is purely imaginary, therefore by taking the imaginary part of the lhs and rhs...

$$f'(x) = \frac{\Im[f(x+i\delta)+0]}{\delta} + \mathcal{O}(\delta)$$

Let's try this approach:

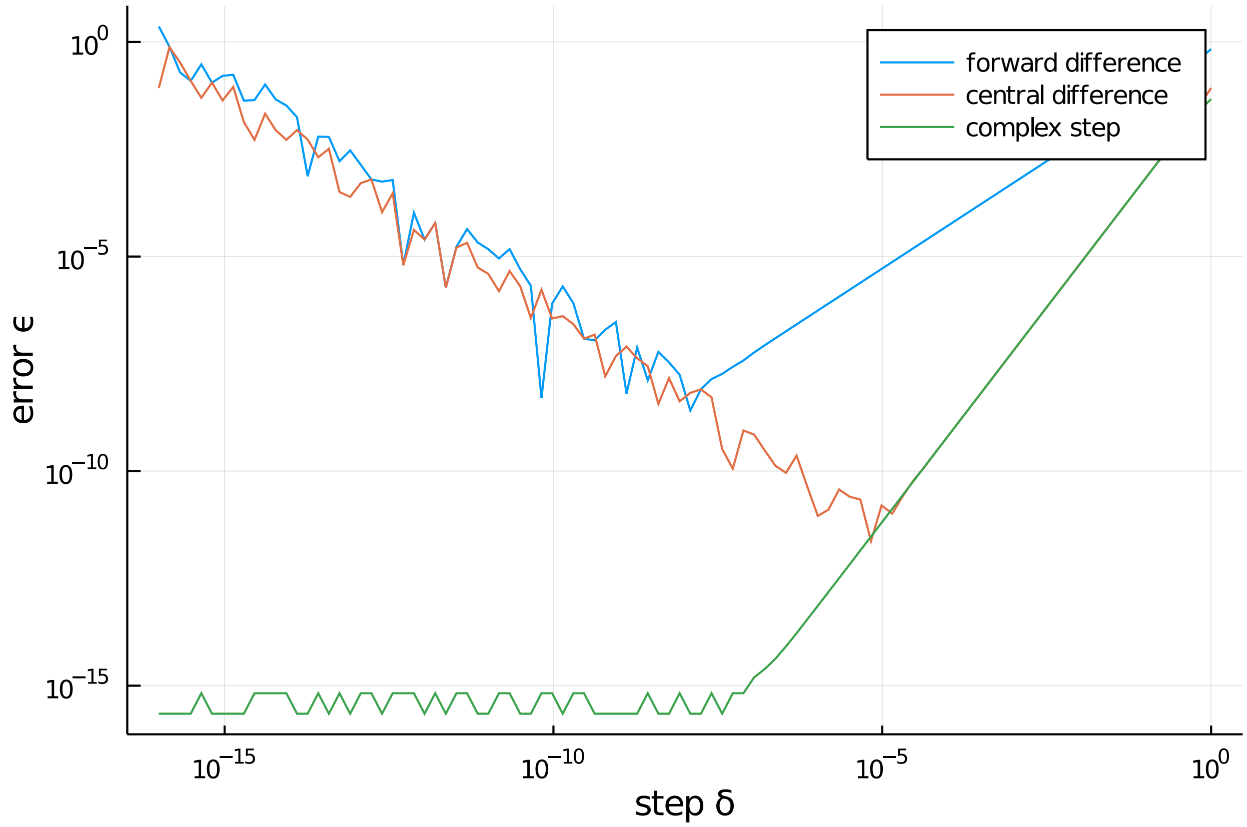
```
complex_difference (generic function with 1 method)
```

```
complex_difference(f, x, δ) = imag(f(x+im*δ)/δ)
```

```
ϵ_cmplx =  Float64[2.22045e-16, 2.22045e-16, 2.22045e-16, 2.22045e-16, 6.66134e-
```

```
ϵ_cmplx = diff_error(complex_difference, x₀, δ_arr) .+ eps(Float64)
```

```
pl3 =
```



```julia
pl3 = plot!(pl2, δ_arr, ϵ_cmplx, xscale=:log10, yscale=:log10, label="complex
step")
```

WOW! practically 0 error!

This is because f is a real function of real inputs, and the step $i\delta$ is purely imaginary, so the real and imaginary part never mix!

No mixing $\rightarrow$ no numerical cancellation and roundoff errors!

Generalizing: Sensitivities and Dual numbers

The derivative can be thought as the sensitivity of a function to it's input:

```
how much does f(x₀) changes when the input x₀ changes by a small amount δ?
```

Thanks to the Taylor's theorem:

$$f(x_0 + \delta) = f(x_0) + \delta f'(x_0) + \dots$$

And now, think about the δ as a component (a bit like the i unit, quaternion directions i, j, k, grassman ϵ...)

$$\mathbb{D}(\mathbb{R}) \sim \mathbb{R} \times \mathbb{R}$$

$$a \in \mathbb{D}(\mathbb{R}) = a_{value} + \delta a_{sensitivity}$$

$$f : \mathbb{D}(\mathbb{R}) \rightarrow \mathbb{D}(\mathbb{R})$$

$$f(a) = f(a) + \delta f'(a)$$

$$a + b = (a_v + b_v) + \delta(a_s + b_s)$$

$$ab = (a_v b_v) + \delta(a_s b_v + a_v b_s)$$

```julia
struct Dual{T}
    val::T
    sen::T
end
```

```julia
Base.:+(f::Dual, g::Dual) = Dual(f.val + g.val, f.sen + g.sen)
```

```julia
Base.:+(f::Dual, α::Number) = Dual(f.val + α, f.sen)
```

```julia
Base.:+(α::Number, f::Dual) = f + α
```

```julia
Base.:*(f::Dual, g::Dual) = Dual(f.val*g.val, f.sen*g.val + f.val*g.sen)
```

```julia
Base.exp(f::Dual) = Dual(exp(f.val), exp(f.val) * f.sen)
```

```julia
Base.log(f::Dual) = Dual(exp(f.val), inv(f.val) * f.sen)
```

```julia
Base.sin(f::Dual) = Dual(sin(f.val), cos(f.val) * f.sen)
```

```
2.3051952967016094
```

```julia
f(Dual(x₀,1.0)).sen
```

```
2.3051952967016094
```

```julia
dfdx(x₀)
```

However, think about the function

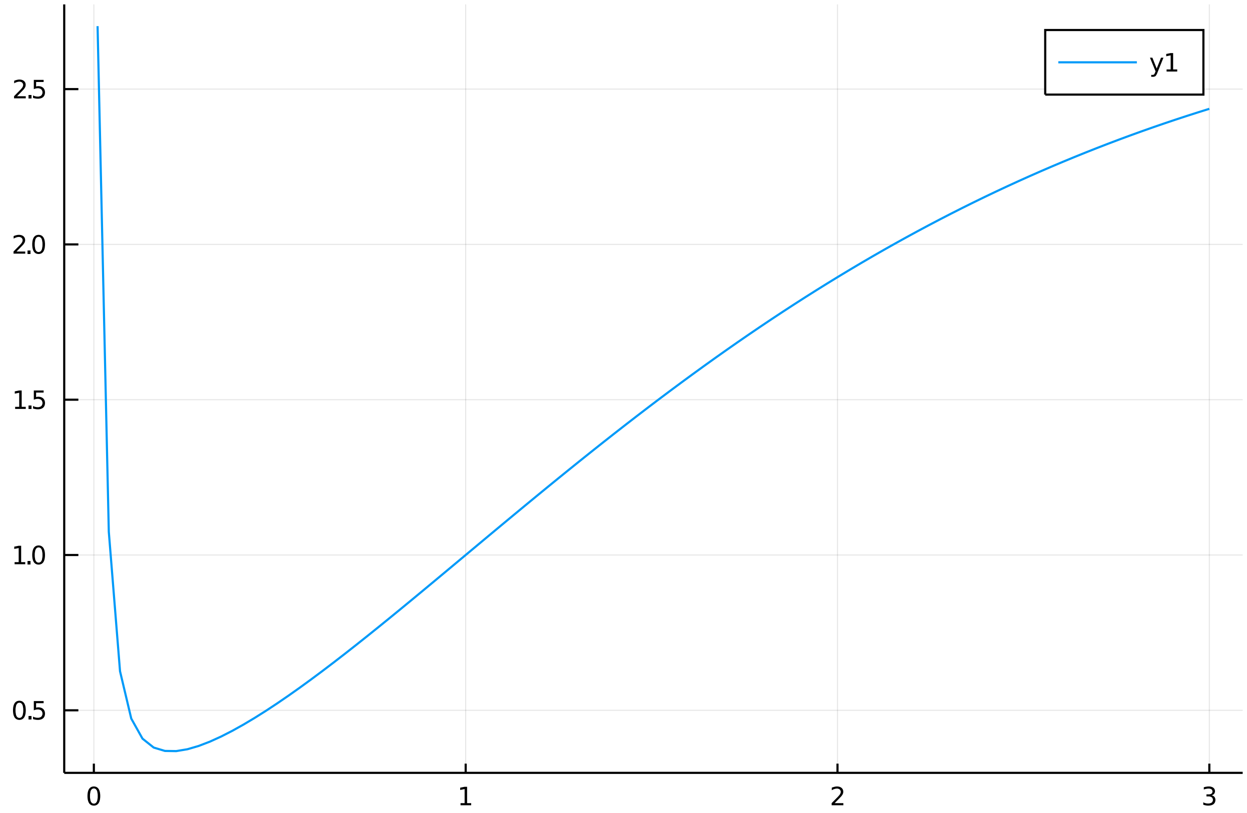
```julia
md"""
However, think about the function
"""
```

An example

A nested function

```
h = #62 (generic function with 1 method)
```

```julia
h = exp ∘ sin ∘ log
```



```
plot(range(0.01, 3.0, length=100), h)
```

```
xᵢ =  Dual{Float64}(0.5,  1.0)
```

```
# Forward mode:
xᵢ = Dual(0.5, 1.0)
```

```
y₁ =  Dual{Float64}(1.64872,  2.0)
```

```
y₁ = log(xᵢ)
```

```
y₂ =  Dual{Float64}(0.996965,  -0.155692)
```

```
y₂ = sin(y₁)
```

```
y₃ =  Dual{Float64}(0.839828,  -0.084518)
```

```
y₃ = sin(y₂)
```