Automatic differentiation

Advanced Topics from Scientific Computing
TU Berlin Winter 2024/25
Notebook 06

(cc) BY-SA **Jürgen Fuhrmann**

```julia
begin
    using CairoMakie, Colors
    CairoMakie.activate!(; type = "svg")
end
```

Contents

Automatic differentiation (AD) has become more and more widely used. It is one of main ingredients of machine learning. This lecture tries to explain some basic ideas on how it works.

For further reading: the Wikipedia article on AD is well resourced.

Reverse mode AD

Let us define a function and see how we can calculate it derivative:

```
f (generic function with 2 methods)
```

```julia
f(x, y) = x^2 + 2x * y + 3y
```

```julia
x = 10.0; y = 11.0;
```

```
353.0
```

```julia
f(x, y)
```

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

```julia
function fdf(x, y)
    #! format: off
    v1 = x^2;     ∂x_v1 = 2x;     ∂y_v1 = 0
    v2 = 2x * y; ∂x_v2 = 2y;     ∂y_v2 = 2x
    v3 = 3y;      ∂x_v3 = 0;      ∂y_v3 = 3

    v4 = v1 + v2 + v3
    ∂x_v4 = ∂x_v1 + ∂x_v2 + ∂x_v3
    ∂y_v4 = ∂y_v1 + ∂y_v2 + ∂y_v3

    v4, ∂x_v4, ∂y_v4
    #! format: on
end
```

```
(353.0, 42.0, 23.0)
```

```julia
fdf(x, y)
```

In order to caculate the derivatives, one essentially parses the expression and recursively applies differentiation rules.

A simple implementation of reverse AD

Here, we provide a simple example how this can be implemented. We modify the implementation by Simeon Schaub in his ReverseModePluto example.

```julia
begin
    using AbstractTrees
    using LinearAlgebra
end
```

Tracked number type

We use the Julia multiple dispatch system in order to design a number type (which we call `Tracked`) which allows to track operations and apply derivative rules.

```julia
begin
    struct Tracked{T} <: Number
        val::T    # value
        name::Symbol # label just for printing
        df::Union{Function, Nothing} # derivative rule combining deps
        deps::Vector{Tracked} # Tracked values of dependent expressions.
    end

    # Construct a tracked value from with optional name from some number
    function Tracked{T}(x, name = gensym()) where {T}
        Tracked{T}(x, name, nothing, Tracked[])
    end

    # Constructor which automatically takes type from number
    Tracked(x::T, name = gensym()) where {T} = Tracked{T}(x, name)
end;
```

Equality, conversion, promotion

When are two `Tracked` instances equal ?

```julia
Base.:(==)(x::Tracked, y::Tracked) = x === y
```

Lift some rules for handling number types to `Tracked`:

Allow to convert tracked values with different type parameters

```julia
function Base.convert(T::Type{Tracked{S}}, x::Tracked) where {S}
    T(convert(S, x.val), x.name, x.df, x.deps)
end
```

Promote rule for handling combinations of tracked values with different type parameters

```julia
function Base.promote_rule(::Type{Tracked{S}},
                           ::Type{T}) where {S <: Number, T <: Number}
    Tracked{promote_type(S, T)}
end
```

Printing tracked numbers as tree

```julia
AbstractTrees.children(x::Tracked) = x.deps
```

```julia
function AbstractTrees.printnode(io::IO, node::Tracked)
    if isnothing(node.df)
        if Base.isgensym(node.name)
            print(io, " ← $(node.val)")
        else
            print(io, "[$(node.name)] ← $(node.val)")
        end
    else
        print(io, "[$(node.name)] = $(node.val)")
    end
end
```

```julia
Base.show(io::IO, ::MIME"text/plain", x::Tracked) = print_tree(io, x)
```

Arithmetics of tracked numbers

Now we can print tracked numbers and define operations using mutiple dispatch for the basic operations on numbers.

```
tx = [x] ← 10.0
```

```julia
tx = Tracked(x, :x)
```

```
ty = [y] ← 11.0
```

```julia
ty = Tracked(y, :y)
```

```julia
function Base.:+(x::Tracked, y::Tracked)
    Tracked(x.val + y.val, :+, Δ -> (Δ, Δ), Tracked[x, y])
end
```

Here, we see the tree for addition

```
[+] = 21.0
├─ [x] ← 10.0
└─ [y] ← 11.0
```

```julia
tx + ty
```

```julia
function Base.:-(x::Tracked, y::Tracked)
    Tracked(x.val - y.val, :-, Δ -> (Δ, -Δ), Tracked[x, y])
end
```

```julia
function Base.:*(x::Tracked, y::Tracked)
    Tracked(x.val * y.val, :*, Δ -> (Δ * y.val, x.val * Δ), Tracked[x, y])
end
```

```julia
function Base.:^(x::Tracked, n::Int)
    Tracked(x.val^n, Symbol("^$n"), Δ -> (Δ * n * x.val^(n - 1),), Tracked[x])
end
```

```julia
function Base.:/(x::Tracked, y::Tracked)
    Tracked(x.val / y.val, :/, Δ -> (Δ / y.val, -Δ * x.val / y.val^2),
            Tracked[x, y])
end
```

These operations are sufficient to be applied to our initial function:

```
[+] = 353.0
├─ [+] = 320.0
│  ├─ [^2] = 100.0
│  │  └─ [x] ← 10.0
│  └─ [*] = 220.0
│     ├─ [*] = 20.0
│     │  ├─  ← 2.0
│     │  └─ [x] ← 10.0
│     └─ [y] ← 11.0
└─ [*] = 33.0
   ├─  ← 3.0
   └─ [y] ← 11.0
```

```julia
f(tx, ty)
```

Collecting the gradient

Set up a dictionary where tracked values are the keys:

```julia
function grad(f::Tracked)
    d = Dict{Any, Any}(f => 1)
    # Depth first tree search (DFS)
    # recursively traverses all dependents, parents before children:
    for x in PreOrderDFS(f)
        # ignore untracked variables like constants
        if x.df === nothing
            continue
        end
        dy = x.df(d[x]) # Evaluate "pullback": parents have been visited before,
        # so they have dictionary entries
        for (yᵢ, dyᵢ) in zip(x.deps, dy)
            # store the gradient in d
            # if we have already stored a gradient for
            # this variable, we need to add them
            if haskey(d, yᵢ)
                d[yᵢ] = d[yᵢ] + dyᵢ
            else
                d[yᵢ] = dyᵢ
            end
        end
    end
    return d
end;
```

```julia
grad(f::Tracked, x::Tracked) = grad(f)[x];
```

```
tf = [+] = 353.0
     ├─ [+] = 320.0
     │  ├─ [^2] = 100.0
     │  │  └─ [x] ← 10.0
     │  └─ [*] = 220.0
     │     ├─ [*] = 20.0
     │     │  ├─ ← 2.0
     │     │  └─ [x] ← 10.0
     │     └─ [y] ← 11.0
     └─ [*] = 33.0
        ├─ ← 3.0
        └─ [y] ← 11.0
```

```julia
tf = f(tx, ty)
```

```
42.0
```

```julia
grad(tf, tx)
```

```
23.0
```

```julia
grad(tf, ty)
```

```julia
function Base.sin(x::Tracked)
    Tracked(sin(x.val), :sin, Δ -> (Δ * cos(x.val),), Tracked[x])
end
```

```julia
function Base.cos(x::Tracked)
    Tracked(cos(x.val), :cos, Δ -> (-Δ * sin(x.val),), Tracked[x])
end
```

```julia
function Base.exp(x::Tracked)
    Tracked(exp(x.val), :exp, Δ -> (Δ * exp(x.val),), Tracked[x])
end
```

So now, we can perform AD for functions:

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

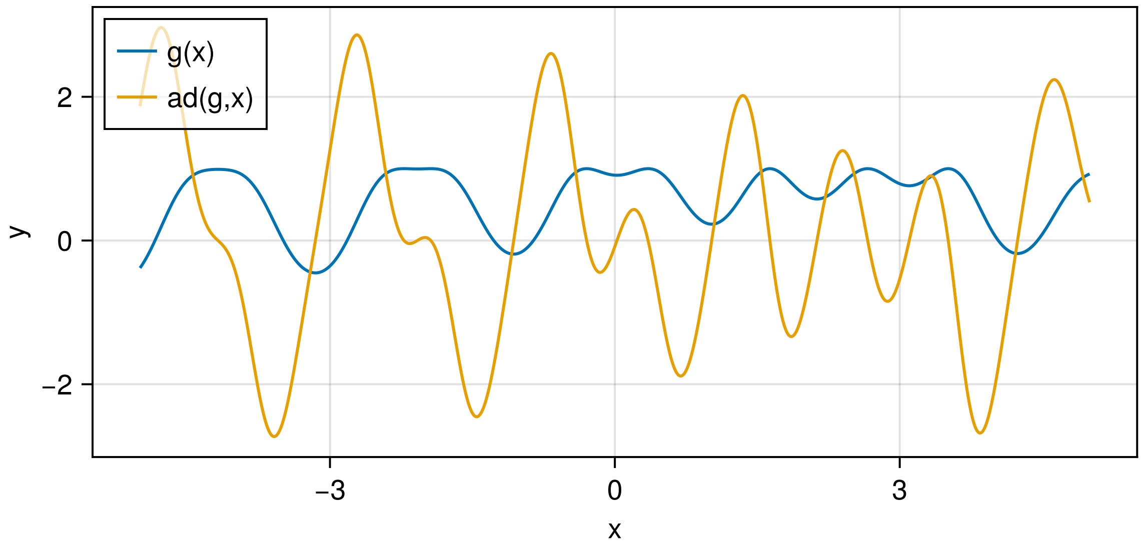
```julia
function ad(f, x)
    tx = Tracked(x)
    grad(f(tx), tx)
end
```

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

```julia
g(x) = sin(exp(0.2 * x) + cos(3x))
```

```
[sin] = 0.9521277180368776
└─ [+] = 7.543307548818235
   ├─ [exp] = 7.38905609893065
   │  └─ [*] = 2.0
   │     ├─ ← 0.2
   │     └─ [x] ← 10.0
   └─ [cos] = 0.15425144988758405
      └─ [*] = 30.0
         ├─ ← 3.0
         └─ [x] ← 10.0
```

```julia
g(tx)
```



```julia
let
    X = (-5:0.01:5)
    fig = Figure(; size = (600, 300))
    axis = Axis(fig[1, 1]; xlabel = "x", ylabel = "y")
    lines!(axis, X, g.(X); label = "g(x)")
    lines!(axis, X, ad.(g, X); label = "ad(g,x)")
    axislegend(axis; backgroundcolor = RGBA(1.0, 1.0, 1.0, 0.7), position = :lt)
    fig
end
```

In general, reverse mode AD implementations use something similar to our Tracked tree, which "records" the function and therefore often is calles a "Tape". This means that in this case AD computation requires to first record the tape, which may result in certain memory overhead.

ReverseDiff.jl and DifferentiationInterface.jl

Julia provides an implementation of reverse mode AD in ReverseDiff.jl and a general interface to AD via DifferentiationInterface.jl

```julia
using ReverseDiff, DifferentiationInterface
```

The AD function needs a vector input, so we define another method for our initial `f`:

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

```julia
f(X::AbstractVector) = f(X[1], X[2])
```

```
(353.0, [42.0, 23.0])
```

```julia
value_and_gradient(f, AutoReverseDiff(), [x, y])
```

Forward mode AD

We cover one particular formulation of forward mode AD based on so-called dual numbers.

Dual numbers

The field of complex numbers $\mathbb{C}$ extends the field of real numbers $\mathbb{R}$ by introducing a number i with $i^2 = -1$.

> *Dual numbers* are defined by extending the real numbers by formally introducing a number ε with $\varepsilon^2 = 0$:
>
> $$\mathbb{D} = \{a + b\varepsilon \mid a, b \in \mathbb{R}\} = \left\{ \begin{pmatrix} a & b \\ 0 & a \end{pmatrix} \mid a, b \in \mathbb{R} \right\} \subset \mathbb{R}^{2\times 2}$$

In contrast to real and complex numbers, dual numbers form a ring, not a field.

Evaluating polynomials on dual numbers: Let $p(x) = \sum_{i=0}^n p_i x^i$. Then

$$\begin{aligned} p(a + b\varepsilon) &= \sum_{i=0}^n p_i a^i + \sum_{i=1}^n i p_i a^{i-1} b\varepsilon \\ &= p(a) + bp'(a)\varepsilon \end{aligned}$$

- This can be generalized to any analytical function. $\Rightarrow$ automatic evaluation of function and derivative at once
- $\Rightarrow$ *forward mode automatic differentiation*
- Multivariate dual numbers: generalization for partial derivatives

Dual numbers in Julia

Nathan Krislock provided a simple dual number arithmetic example in Julia.

- Define a struct parametrized with type T. This is akin a template class in C++
- The type shall work with all methods working with `Number`
- In order to construct a Dual number from arguments of different types, allow promotion aka "parameter type homogenization"

```julia
begin
    struct DualNumber{T} <: Number where {T <: Real}
        value::T
        deriv::T
    end
    DualNumber(v, d) = DualNumber(promote(v, d)...)
end;
```

Define a way to convert a `Real` to `DualNumber`

```julia
function Base.promote_rule(::Type{DualNumber{T}},
                           ::Type{<:Real}) where {T <: Real}
    DualNumber{T}
end
```

```julia
function Base.convert(::Type{DualNumber{T}}, x::Real) where {T <: Real}
    DualNumber(x, zero(T))
end
```

Constructing a dual number:

```
d = DualNumber(5, 4)
```

```julia
d = DualNumber(5, 4)
```

Accessing its components:

```
(5, 4)
```

```julia
d.value, d.deriv
```

Simple arithmetic for dual numbers:

Add methods to the functions `+`, `/`, `*`, `-`, `inv` which allow them to work for `DualNumber`

```julia
begin
    import Base: +, /, *, -, inv
    function +(x::DualNumber, y::DualNumber)
        DualNumber(x.value + y.value, x.deriv + y.deriv)
    end

    -(y::DualNumber) = DualNumber(-y.value, -y.deriv)

    -(x::DualNumber, y::DualNumber) = x + -y

    function *(x::DualNumber, y::DualNumber)
        DualNumber(x.value * y.value, x.value * y.deriv + x.deriv * y.value)
    end

    function inv(y::DualNumber{T}) where {T <: Union{Integer, Rational}}
        DualNumber(1 // y.value, (-y.deriv) // y.value^2)
    end

    function inv(y::DualNumber{T}) where {T <: Union{AbstractFloat,
                                                      AbstractIrrational}}
        DualNumber(1 / y.value, (-y.deriv) / y.value^2)
    end

    /(x::DualNumber, y::DualNumber) = x * inv(y)
end;
```

```julia
function Base.sin(x::DualNumber{T}) where {T}
    DualNumber(sin(x.value), cos(x.value) * x.deriv)
end;
```

```julia
function Base.log(x::DualNumber{T}) where {T}
    DualNumber(log(x.value), x.deriv / x.value)
end
```

Define a function for comparison with known derivative:

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

```julia
function testdual(x, f, df)
    xdual = DualNumber(x, 1)
    fdual = f(xdual)
    _f = f(x)
    _df = df(x)
    err = _df - fdual.deriv
    (f = _f, f_dual = fdual.value), (df = _df, df_dual = fdual.deriv),
    (error = err,)
end
```

Polynomial expressions:

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

```julia
p(x) = x^3 + 2x + 1
```

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

```julia
dp(x) = 3x^2 + 2
```

```
((f = 34, f_dual = 34), (df = 29, df_dual = 29), (error = 0))
```

```julia
testdual(3, p, dp)
```

Standard functions:

```
((f = 0.420167, f_dual = 0.420167), (df = 0.907447, df_dual = 0.907447), (error = 0.0))
```

```julia
testdual(13, sin, cos)
```

```
((f = 2.56495, f_dual = 2.56495), (df = 0.0769231, df_dual = 0.0769231), (error = 0.0))
```

```julia
testdual(13, log, x -> 1 / x)
```

Function composition:

```
((f = -0.506366, f_dual = -0.506366), (df = 17.2464, df_dual = 17.2464), (error = 0.0))
```

```julia
testdual(10, x -> sin(x^2), x -> 2x * cos(x^2))
```

If we apply dual numbers in the right way, we can do calculations with derivatives of complicated nonlinear expressions without the need to write code to calculate derivatives.

ForwardDiff.jl

```julia
using ForwardDiff
```

The ForwardDiff.jl package provides a full implementation of these facilities.

```
testdual1 (generic function with 1 method)
```

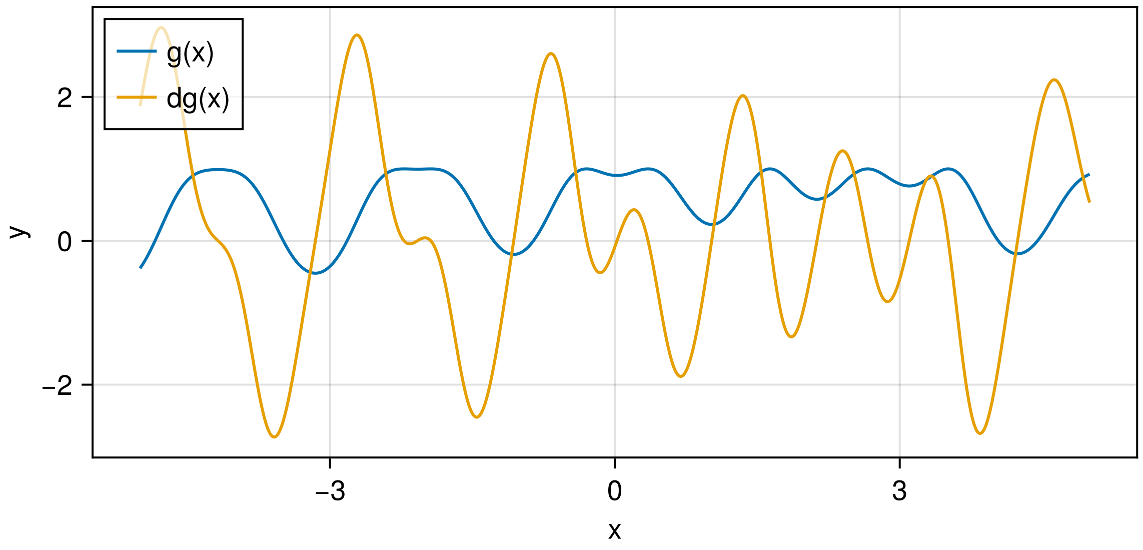
```julia
function testdual1(x, f, df)
    _df = df(x)
    _df_dual = ForwardDiff.derivative(f, x)
    (f = f(x), df = _df, df_dual = _df_dual, error = abs(_df - _df_dual))
end
```

```
(f = 0.14112, df = -0.989992, df_dual = -0.989992, error = 0.0)
```

```julia
testdual1(3, sin, cos)
```

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

```julia
dg(x) = ForwardDiff.derivative(g, x)
```



```julia
let
    X = (-5:0.01:5)
    fig = Figure(; size = (600, 300))
    axis = Axis(fig[1, 1]; xlabel = "x", ylabel = "y")
    lines!(axis, X, g.(X); label = "g(x)")
    lines!(axis, X, dg.(X); label = "dg(x)")
    axislegend(axis; backgroundcolor = RGBA(1.0, 1.0, 1.0, 0.7), position = :lt)
    fig
end
```

It is also possible to use ForwardDiff with DifferentiationInterface.

```
(353.0, [42.0, 23.0])
```

```julia
value_and_gradient(f, AutoForwardDiff(), [x, y])
```

Forward or Reverse ?

Generally, for the calculation of the Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^m$, forward mode AD is more efficient if $n << m$ and reverse mode is more efficient if $n >> m$. The later is important for the calculation of the gradients of the loss functions with respect to the weights in neural network training.

In this course we will stick to the former, as the Jacobian calculation for PDE discretization can be subdivided into operations e.g. on each finite element which have a small number `n` of input variables.

Solving nonlinear systems of equations

Let $A_1 \dots A_n$ be functions depending on n unknowns $u_1 \dots u_n$. Solve the system of nonlinear equations:

$$A(u) = \begin{pmatrix} A_1(u_1 \dots u_n) \\ A_2(u_1 \dots u_n) \\ \vdots \\ A_n(u_1 \dots u_n) \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix} = f$$

$A(u)$ can be seen as a nonlinar operator $A : D \to \mathbb{R}^n$ where $D \subset \mathbb{R}^n$ is its domain of definition.

There is no analogon to Gaussian elimination, so we need to solve iteratively.

Fixpoint iteration scheme

Assume $A(u) = M(u)u$ where for each u, $M(u) : \mathbb{R}^n \to \mathbb{R}^n$ is a linear operator.

Then we can define the iteration scheme: choose an initial value u_0 and at each iteration step, solve

$$M(u^i)u^{i+1} = f$$

Terminate if

$$||A(u^i) - f|| < \varepsilon \quad \text{(residual based)}$$

or

$$||u_{i+1} - u_i|| < \varepsilon \quad \text{(update based)}.$$

- Large domain of convergence

- Convergence may be slow
- Smooth coefficients not necessary

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

```julia
function fixpoint!(u, M, f; imax = 100, tol = 1.0e-10)
    history = Float64[]
    for i = 1:imax
        res = norm(M(u) * u - f)
        push!(history, res)
        if res < tol
            return u, history
        end
        u = M(u) \ f
    end
    error("No convergence after $imax iterations")
end
```

Definition of M(u)

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

```julia
function M(u)
    [1+1.2 * (u[1]^2 + u[2]^2) -(u[1]^2 + u[2]^2);
     -(u[1]^2 + u[2]^2) 1+1 * (u[1]^2 + u[2]^2)]
end
```

```
F =  [1, 3]
```

```julia
F = [1, 3]
```

```
([1.28822, 1.61348], [3.16228, 26.9072, 1.45019, 1.87735, 0.614397, 0.471544, 0.229973,
```

```julia
fixpt_result, fixpt_history = fixpoint!([0, 0], M, F; imax = 1000,
                                         tol = 1.0e-10)
```

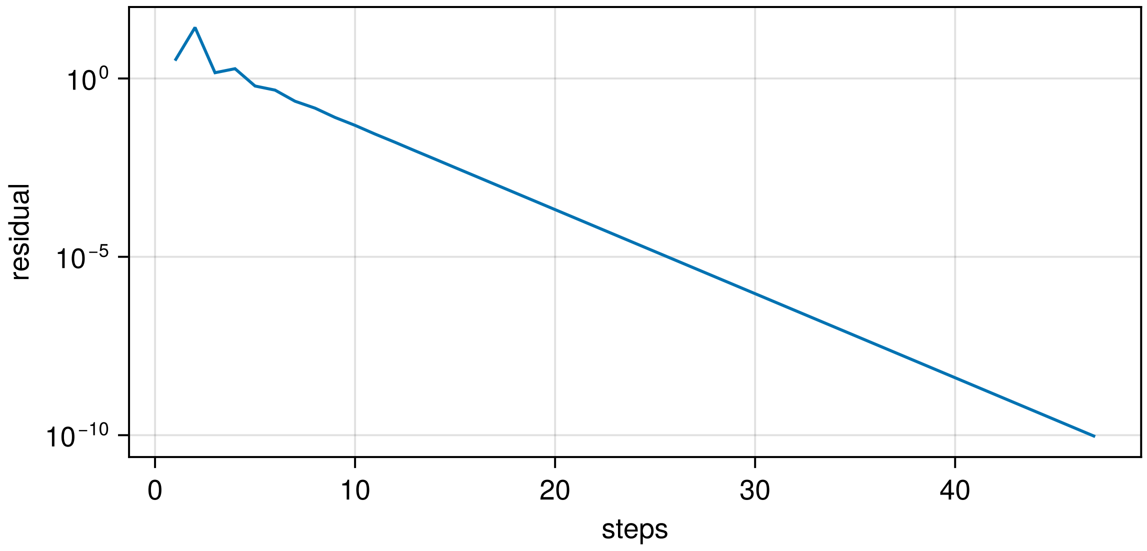
```
contraction (generic function with 1 method)
```

```julia
contraction(h) = h[2:end] ./ h[1:(end - 1)]
```

```julia
function plothistory(history::Vector{<:Number})
    fig = Figure(; size = (600, 300))
    axis = Axis(fig[1, 1]; xlabel = "steps", ylabel = "residual",
                yscale = log10)
    lines!(axis, 1:length(history), history)
    #    axislegend(axis,bgcolor = RGBA(1.0, 1.0, 1.0, 0.7),position=:lt)
    fig
end;
```

```
[8.50882, 0.0538958, 1.29456, 0.327268, 0.76749, 0.487702, 0.640077, 0.548586, 0.60068
```

```julia
contraction(fixpt_history)
```



```
plothistory(fixpt_history)
```

```
[1.85807e-11, -8.93863e-11]
```

```
M(fixpt_result) * fixpt_result - F
```

Newton iteration scheme

The fixed point iteration scheme assumes a particular structure of the nonlinear system. In addition, one would need to investigate convergence conditions for each particular operator. Can we do better ?

Let $A'(u)$ be the *Jacobi matrix* of first partial derivatives of A at point u:

$$A'(u) = (a_{kl})$$

'with

$$a_{kl} = \frac{\partial}{\partial u_l} A_k(u_1 \dots u_n)$$

Then, one calculates in the i-th iteration step:

$$u_{i+1} = u_i - (A'(u_i))^{-1}(A(u_i) - f)$$

One can split this a follows:

- Calculate residual: $r_i = A(u_i) - f$
- Solve linear system for update: $A'(u_i)h_i = r_i$
- Update solution: $u_{i+1} = u_i - h_i$

General properties are:

- Potenially small domain of convergence - one needs a good initial value
- Possibly slow initial convergence

- Quadratic convergence close to the solution

Linear and quadratic convergence

Let $e_i = u_i - \hat{u}$.

- Linear convergence: observed for e.g. linear systems: Asymptotically constant error contraction rate

$$\frac{||e_{i+1}||}{||e_i||} \sim \rho < 1$$

- Quadratic convergence: $\exists i_0 > 0$ such that $\forall i > i_0$, $\frac{||e_{i+1}||}{||e_i||^2} \leq M < 1.$
    - As $||e_i||$ decreases, the contraction rate decreases:

$$\frac{\frac{||e_{i+1}||}{||e_i||}}{\frac{||e_i||}{||e_{i-1}||}} = \frac{||e_{i+1}||}{\frac{||e_i||^2}{||e_{i-1}||}} \leq ||e_{i-1}||M$$

- In practice, we can watch $||r_i||$ or $||h_i||$

Newton method with AD

This is the situation where we could apply automatic differentiation for vector functions of vectors.

```
A1 (generic function with 1 method)
```

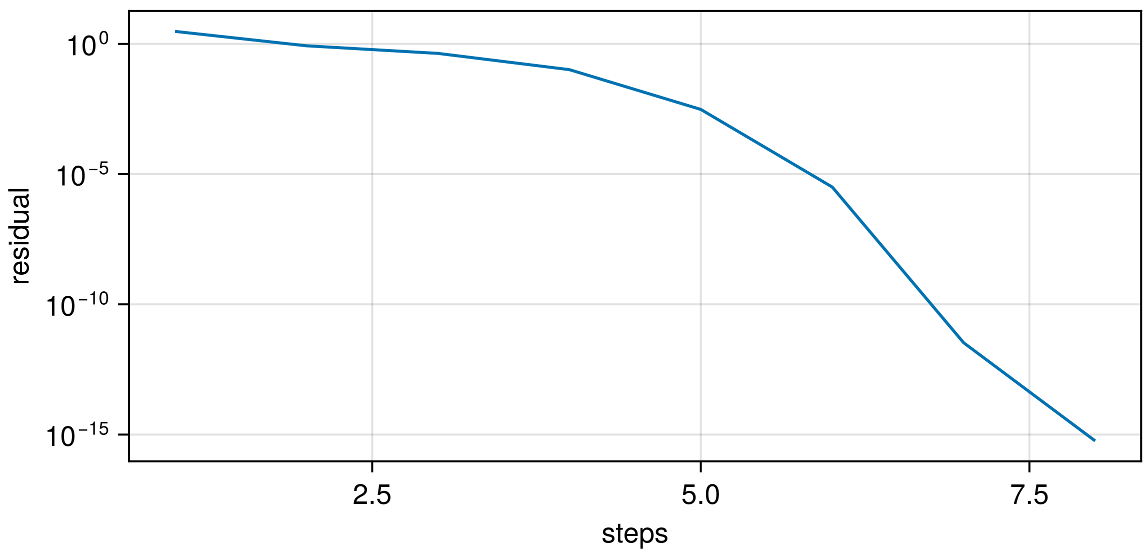
```
A1(u) = M(u) * u
```

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

```julia
function newton(A, b, u0; tol = 1.0e-12, maxit = 100)
    history = Float64[]
    u = copy(u0)
    it = 0
    converged = false
    while !converged && it < maxit
        res = A(u) - b
        jac = ForwardDiff.jacobian((v) -> A(v) - b, u)
        h = jac \ res
        u -= h
        nm = norm(h)
        push!(history, nm)
        it = it + 1
        if nm < tol
            converged = true
        end
    end
    if converged
        return u, history
    else
        throw("convergence failed")
    end
end
```

```
([1.28822, 1.61348], [3.02185, 0.846373, 0.432681, 0.102853, 0.0030576, 3.19945e-6, 3.3
```

```julia
newton_result1, newton_history1 = newton(A1, F, [0, 0.1]; tol = 1.e-13)
```



```julia
plothistory(newton_history1)
```

Calculate function and derivative at once ?

Let us take a more complicated example with an operator dependent on a parameter λ which allows to adjust the "severity" of the nonlinearity. For λ=0, it is linear, for λ=1 it is strongly nonlinear.

```
A2λ (generic function with 1 method)
```

```julia
function A2λ(x, λ)
    [x[1] + 10λ * x[1]^5 + 3λ * x[2] * x[3],
     0.1 * x[2] + 10λ * x[2]^5 - 3λ * x[1] - x[3],
     10λ * x[3]^5 + 10λ * x[1] * x[2] * x[3] + x[3] / 100]
end
```

```
A2 (generic function with 1 method)
```

```julia
A2(x) = A2λ(x, 1)
```

```
F2 =  [0.1, 0.1, 0.1]
```

```julia
F2 = [0.1, 0.1, 0.1]
```

```
U02 =  [1.0, 1.0, 1.0]
```

```julia
U02 = [1, 1.0, 1.0]
```

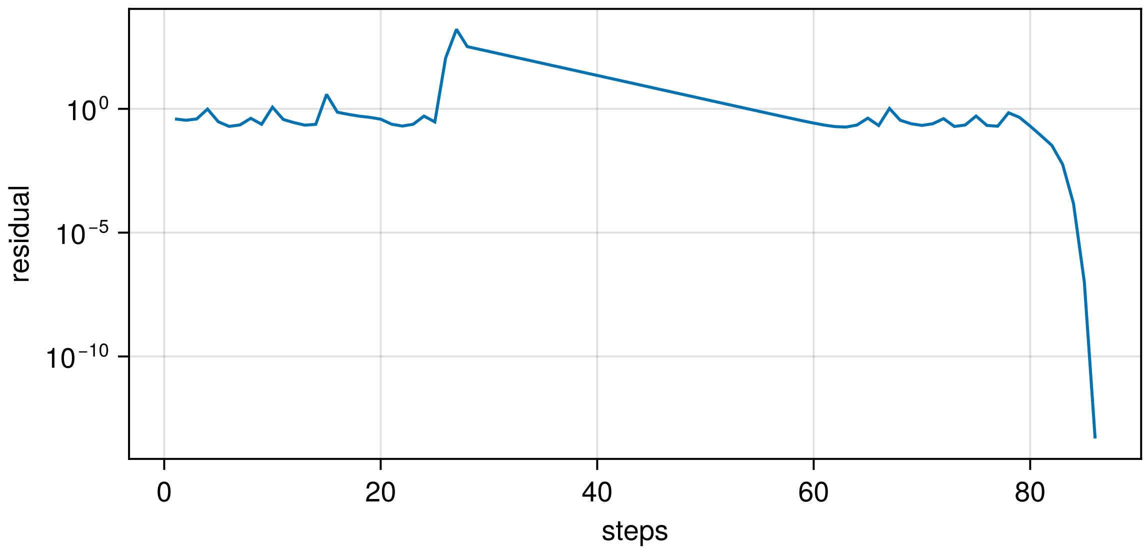
```
([-0.188484, 0.198519, 0.488388], [0.39077, 0.345694, 0.389908, 0.977557, 0.300465, 0.1
```

```julia
res2, hist2 = newton(A2, F2, U02)
```

```
[0.0, 8.32667e-17, -5.55112e-17]
```

```julia
A2(res2) - F2
```

Newton steps: 86



```julia
plothistory(hist2)
```

Here, we observe that we have to use lots of iteration steps and see a rather erratic behaviour of the residual. After ≈ 80 steps we arrive in the quadratic convergence region where convergence is fast.

Damped Newton scheme

There are may ways to improve the convergence behaviour and/or to increase the convergence radius in such a case. The simplest ones are:

- find a good estimate of the initial value
- damping: do not use the full update, but damp it by some factor which we increase during the iteration process until it reaches 1

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

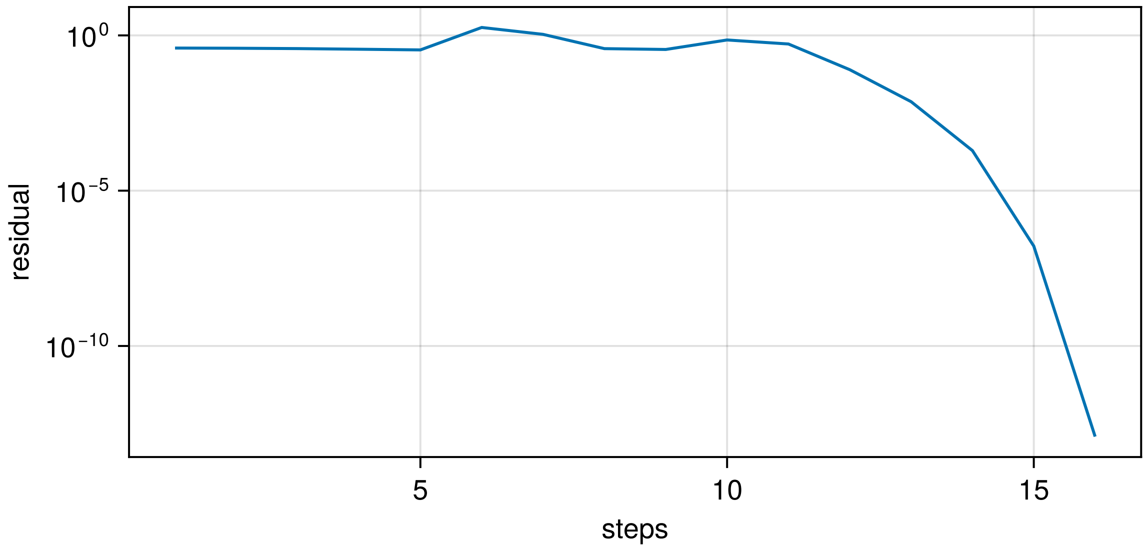
```julia
function dnewton(A, b, u0; tol = 1.0e-12, maxit = 100, damp = 1,
                 damp_growth = 1)
    result = DiffResults.JacobianResult(u0)
    history = Float64[]
    u = copy(u0)
    it = 1
    while it < maxit
        ForwardDiff.jacobian!(result, (v) -> A(v) - b, u)
        res = DiffResults.value(result)
        jac = DiffResults.jacobian(result)
        h = jac \ res
        u .-= damp * h
        nm = norm(h)
        push!(history, nm)
        if nm < tol
            return u, history
        end

        it = it + 1
        damp = min(damp * damp_growth, 1.0)
    end
    throw("convergence failed")
end
```

In this implementation, we also try to save work by evaluating result and Jacobian once.

```
([-0.188484, 0.198519, 0.488388], [0.39077, 0.38541, 0.375394, 0.358292, 0.340649, 1.79
```

```julia
res3, hist3 = dnewton(A2, F2, U02; damp = 0.1, damp_growth = 2, maxit = 1000)
```

Newton steps: 16



```
plothistory(hist3)
```

```
[-2.77556e-17, -2.77556e-17, 0.0]
```

```
A2(res3) - F2
```

The example shows: damping indeed helps to improve the convergece behaviour. If we would keep the damping parameter less than 1, we loose the quadratic convergence behavior.

A more sophisticated strategy would be line search: automatic detection of a damping factor which prevents the residual from increasing.

Parameter embedding

Another option is the use of parameter embedding for parameter dependent problems.

- Problem: solve $A(u_\lambda, \lambda) = f$ for $\lambda = 1$.
- Assume $A(u_0, 0)$ can be easily solved.
- Choose step size δ

1. Solve $A(u_0, 0) = f$
2. Set $\lambda = 0$
3. Solve $A(u_{\lambda+\delta}, \lambda + \delta) = f$ with initial value u_λ
4. Set $\lambda = \lambda + \delta$
5. If $\lambda < 1$ repeat with 3.

- If δ is small enough, we can ensure that u_λ is a good initial value for $u_{\lambda+\delta}$.
- Possibility to adapt δ depending on Newton convergence

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

```julia
function embed_newton(A, F, U0; δ0 = 0.1, δgrowth = 1.2, λ0 = 0, λ1 = 1)
    U = copy(U0)
    allhist = Vector[]
    λ = λ0
    δ = δ0
    while true
        U, hist = newton(x -> A(x, λ), F, U)
        push!(allhist, hist)
        if λ == λ1
            break
        end
        λ = min(λ + δ, λ1)
        δ *= δgrowth
    end
    U, allhist
end
```

```
([-0.188484, 0.198519, 0.488388], [[100.408, 1.41554e-14], [28.0258, 16.6762, 13.3379,
```

```julia
res4, hist4 = embed_newton(A2λ, F2, U02; δ0 = 0.01, δgrowth = 5.0)
```

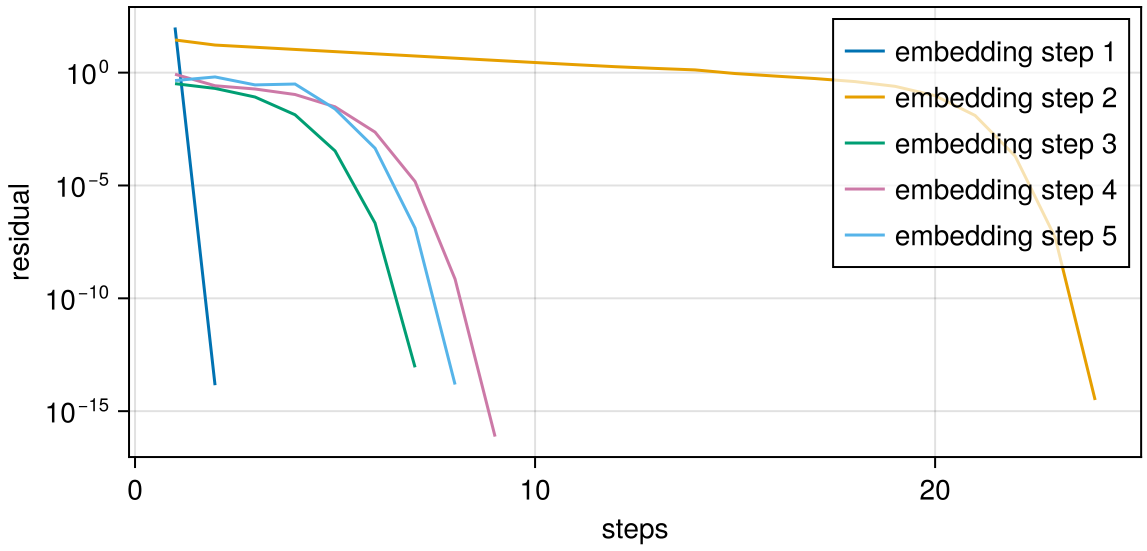
```
[0.0, 8.32667e-17, -5.55112e-17]
```

```julia
A2λ(res4, 1.0) - F2
```

Newton steps: 50

```
plothistory (generic function with 2 methods)
```

```julia
function plothistory(allhistory::Vector{<:AbstractVector})
    fig = Figure(; size = (600, 300))
    axis = Axis(fig[1, 1]; xlabel = "steps", ylabel = "residual",
                yscale = log10)
    i = 1
    for history in allhistory
        lines!(1:length(history), history; label = "embedding step $(i)")
        i = i + 1
    end
    axislegend(axis; backgroundcolor = RGBA(1.0, 1.0, 1.0, 0.7), position = :rt)
    fig
end
```



```julia
plothistory(hist4)
```

NLsolve.jl

```julia
using NLsolve
```

```
nlres1 = Results of Nonlinear Solver Algorithm
          * Algorithm: Trust-region with dogleg and autoscaling
          * Starting Point: [1.0, 1.0, 1.0]
          * Zero: [0.0575824474961491, 0.48399543065156675, 0.04126490344485912]
          * Inf-norm of residuals: 0.088086
          * Iterations: 1000
          * Convergence: false
            * |x - x'| < 0.0e+00: false
            * |f(x)| < 1.0e-08: false
          * Function Calls (f): 74
          * Jacobian Calls (df/dx): 37
```

```julia
nlres1 = nlsolve(u -> A2λ(u, 1.0) - F2, U02)
```

```
[0.0175049, -2.60121e-5, -0.0880858]
```

```julia
A2λ(nlres1.zero, 1.0) - F2
```

```
nlres2 = Results of Nonlinear Solver Algorithm
          * Algorithm: Newton with line-search
          * Starting Point: [1.0, 1.0, 1.0]
          * Zero: [-0.18848435787886608, 0.19851914493466086, 0.48838826112819433]
          * Inf-norm of residuals: 0.000000
          * Iterations: 87
          * Convergence: true
            * |x - x'| < 0.0e+00: false
            * |f(x)| < 1.0e-08: true
          * Function Calls (f): 88
          * Jacobian Calls (df/dx): 88
```

```julia
nlres2 = nlsolve(u -> A2λ(u, 1.0) - F2, U02; method = :newton)
```

```
[-5.3163e-12, 3.84554e-13, 6.42428e-11]
```

```julia
A2λ(nlres2.zero, 1.0) - F2
```

```
nlres3 = Results of Nonlinear Solver Algorithm
          * Algorithm: Newton with line-search
          * Starting Point: [1.0, 1.0, 1.0]
          * Zero: [-0.18848435786937284, 0.19851914494226675, 0.48838826110144995]
          * Inf-norm of residuals: 0.000000
          * Iterations: 85
          * Convergence: true
            * |x - x'| < 0.0e+00: false
            * |f(x)| < 1.0e-08: true
          * Function Calls (f): 86
          * Jacobian Calls (df/dx): 86
```

```
nlres3 = nlsolve(u -> A2λ(u, 1.0) - F2, U02; method = :newton,
                 autodiff = :forward)
```

```
[-7.93809e-15, 4.16334e-16, 1.06332e-13]
```

```
A2λ(nlres3.zero, 1.0) - F2
```

Summary

- Automatic differentiation significantly simplifies handling of nonlinear systems
- Newton method with increasing damping + update based convergence control is rather robust - I use this in my everyday work
- Additional parameter embedding can help to solve even strongly nonlinear problems
- NLSolve.jl provides a convenient default first stop for solving nonlinear systems in Julia, it relies on a number of peer reviewed strategies

---

```
using HypertextLiteral, PlutoUI
```