Scientific Computing TU Berlin Winter 2021/22 © Jürgen Fuhrmann
Notebook 25

```
begin
    ENV["LANG"]="C"
    using PlutoUI
    using PyPlot
    using LinearAlgebra
    using ForwardDiff
    using DiffResults
    PyPlot.svg(true)
end;
```

Contents

Nonlinear systems of equations

Automatic differentiation

Dual numbers

We all know the field of complex numbers $\mathbb{C}$: they extend the real numbers $\mathbb{R}$ based on the introduction of i with $i^2 = -1$.

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

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

A custom dual number type

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"

```
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`

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

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

```

Simple arithmetic for dual numbers:

All these definitions add methods to the functions `+, /, *, -, inv` which allow them to work for `DualNumber`

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

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

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

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

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

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

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

```

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

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

Constructing a dual number:

```
d =  DualNumber(2, 1)
```

```
d=DualNumber(2,1)
```

Accessing its components:

```
(2, 1)
```

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

Define a function for comparison with known derivative:

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

```
function testdual(x,f,df)
    xdual=DualNumber(x,1)
    fdual=f(xdual)
    (f=f(x),f_dual=fdual.value),(df=df(x),df_dual=fdual.deriv)
end
```

Polynomial expressions:

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

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

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

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

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

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

Standard functions:

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

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

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

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

Function composition:

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

```
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

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

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

```
function testdual1(x,f,df)
    (f=f(x),df=df(x),df_dual=ForwardDiff.derivative(f,x))
end
```

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

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

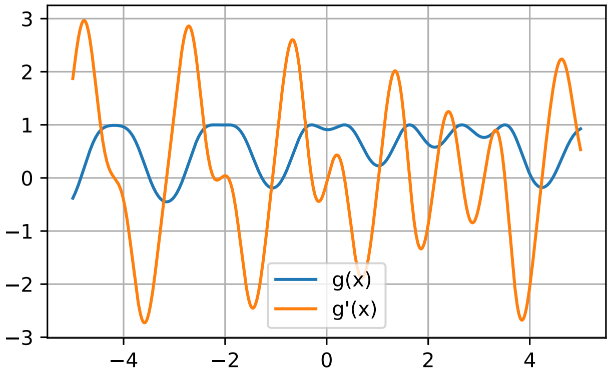
Let us plot some complicated function:

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

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

```
X = -5.0:0.01:5.0
```

```
X=(-5:0.01:5)
```




```
let
    clf()
    grid()
    plot(X,g.(X),label="g(x)")
    plot(X,ForwardDiff.derivative.(g,X), label="g'(x)")
    legend()
    gcf().set_size_inches(5,3)
    gcf()
end
```

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)
```

```
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
```

Example problem

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

```
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]
```

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

```
([1.28822, 1.61348], [3.16228, 26.9072, 1.45019, 1.87735, 0.614397, 0.471544, 0.229973, 0.
```

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

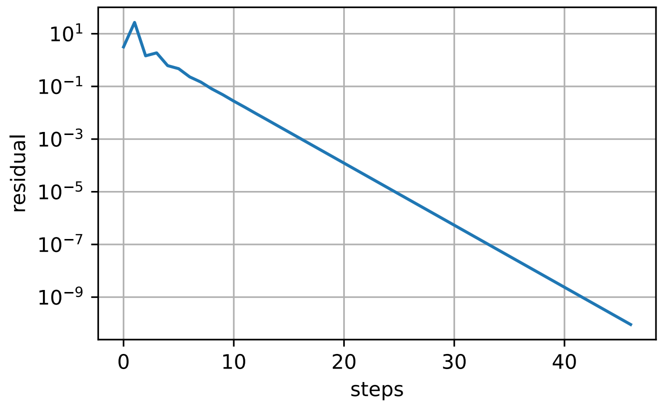
```
contraction (generic function with 1 method)
```

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

```
function plothistory(history::Vector{<:Number})
    clf()
    semilogy(history)
    xlabel("steps")
    ylabel("residual")
    grid()
    gcf()
end;
```

```
[8.50882, 0.0538958, 1.29456, 0.327268, 0.76749, 0.487702, 0.640077, 0.548586, 0.60068, 0.
```

```
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 $||h_i||$

Automatic differentiation for Newton's method

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

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

```
A(u)=M(u)*u
```

Create a result buffer for $n = 2$

```
dresult =
MutableDiffResult([6.8994571147445e-310, 0.0], ([6.8996392119662e-310 6.899515458974e-310;
```

```
dresult=DiffResults.JacobianResult(ones(2))
```

Calculate function and derivative at once:

```
MutableDiffResult([5.199999999999999, 2.0], ([12.2 -6.4; -8.0 9.0],))
```

```
ForwardDiff.jacobian!(dresult,A,[2.0, 2.0])
```

```
[5.2, 2.0]
```

```
DiffResults.value(dresult)
```

```
2×2 Matrix{Float64}:
 12.2  -6.4
 -8.0   9.0
```

```
DiffResults.jacobian(dresult)
```

A Newton solver with automatic differentiation

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

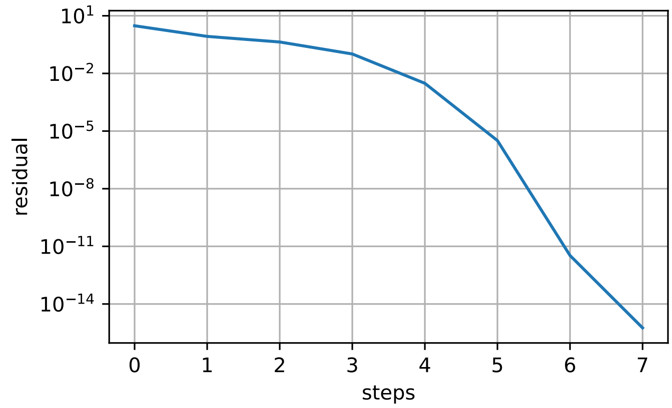
```
function newton(A,b,u0; tol=1.0e-12, maxit=100)
    result=DiffResults.JacobianResult(u0)
    history=Float64[]
    u=copy(u0)
    it=1
    while it<maxit
        ForwardDiff.jacobian!(result,(v)->A(v)-b ,u)
        res=DiffResults.value(result)
        jac=DiffResults.jacobian(result)
        h=jac\res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

        it=it+1
    end
    throw("convergence failed")
end
```

```
([1.28822, 1.61348], [3.02185, 0.846373, 0.432681, 0.102853, 0.0030576, 3.19945e-6, 3.3511
```

```
newton_result,newton_history=newton(A,F,[0,0.1],tol=1.e-13)
```

```
[0.280085, 0.511218, 0.237711, 0.0297278, 0.00104639, 1.04742e-6, 0.000170942]
```

```
contraction(newton_history)
```




```
plothistory(newton_history)
```

```
[8.88178e-16, 8.88178e-16]
```

```
A(newton_result)-F
```

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)
```

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

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

```
A2(x)=A2λ(x,1)
```

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

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

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

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

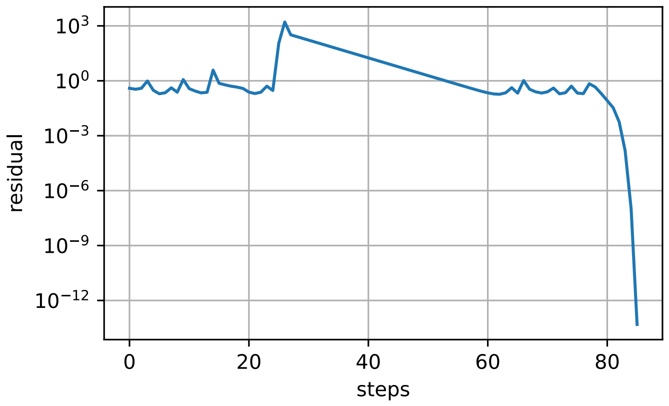
```
([-0.188484, 0.198519, 0.488388], [0.39077, 0.345694, 0.389908, 0.977557, 0.300465, 0.1952
```

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

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

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

Newton steps: 86




```
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 iteration

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

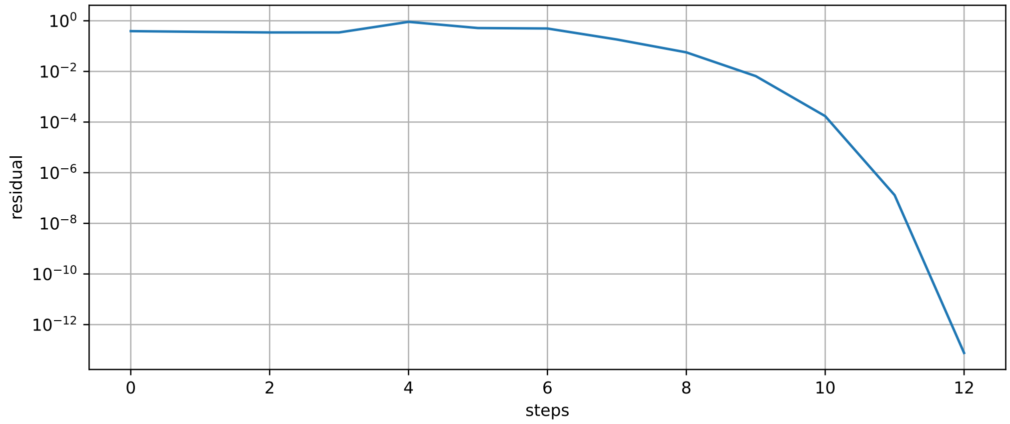
```
dnewton (generic function with 1 method)
```

```
function dnewton(A,b,u0; tol=1.0e-12,maxit=100,damp=0.01,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
```

```
([-0.188484, 0.198519, 0.488388], [0.39077, 0.365701, 0.34648, 0.347045, 0.908798, 0.51711
```

```
res3,hist3=dnewton(A2,F2,U02,damp=0.5,damp_growth=1.1)
```

Newton steps: 13




```
plothistory(hist3)
```

```
[2.77556e-17, -1.38778e-16, 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)
```

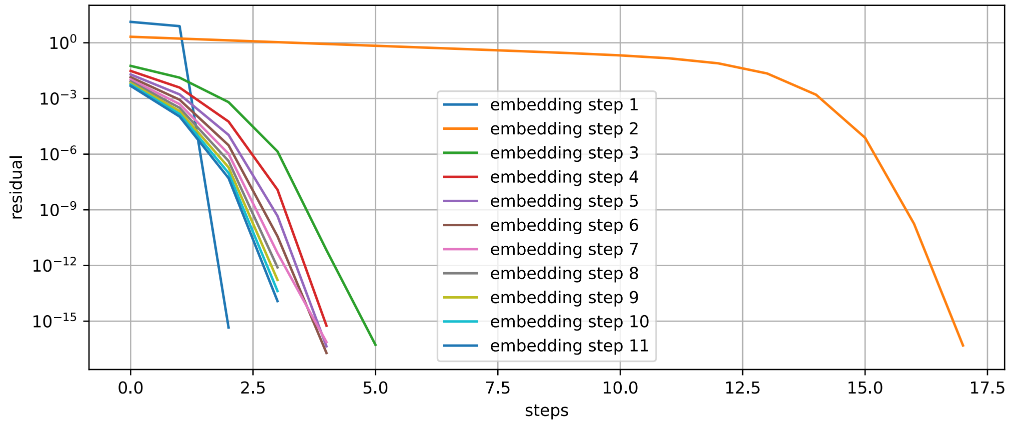
```
function embed_newton(A,F,U0; δ=0.1, λ0=0,λ1=1)
U=copy(U0)
allhist=Vector[]
for λ=λ0:δ:λ1
  U,hist=newton(x->A(x,λ),F,U)
    push!(allhist,hist)
end
   U,allhist
end
```

```
([-0.188484, 0.198519, 0.488388], [[13.3828, 7.87804, 4.57156e-16], [2.11017, 1.68647, 1.3
```

```
res4,hist4=embed_newton(A2λ,F2,U02,δ=0.1,λ0=0)
```

Newton steps: 63

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




```
plothistory(hist4)
```

- Parameter embedding + damping + update based convergence control go a long way to solve even strongly nonlinear problems!
- A similar approach can be used for time dependent problems.