Advanced Topics from Scientific Computing
TU Berlin Winter 2022/23
Notebook 06
Jürgen Fuhrmann

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

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))
```

Constructing a dual number:

d = DualNumber(5, 4)

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

Accessing its components:

(5, 4)

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

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)
```

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)
    _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)

```
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), (error = 0))

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

Standard functions:

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

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

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

```
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))

```
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)
    _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)

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

Let us plot some complicated function:

g (generic function with 1 method)

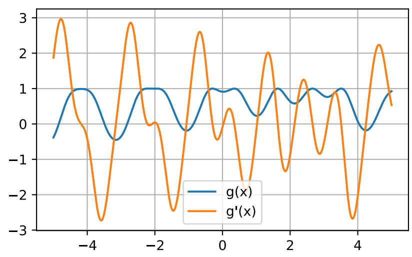
```
g(x)=sin(exp(0.2*x)+cos(3x))
```

dg (generic function with 1 method)

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

X = -5.0:0.01:5.0

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



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
```

Definition of M(u)

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:  [1.28822, 1.61348]
    2:  [3.16228, 26.9072, 1.45019, 1.87735, 0.614397, 0.471544, 0.229973, 0.1472, 0.0807!
)
```

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

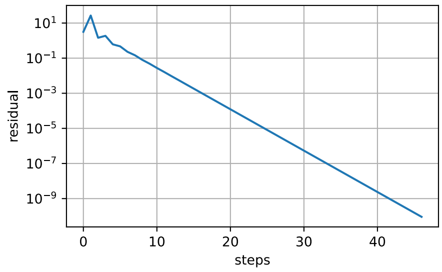
contraction (generic function with 1 method)

```
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 $||h_i||$

newton1: 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)
```

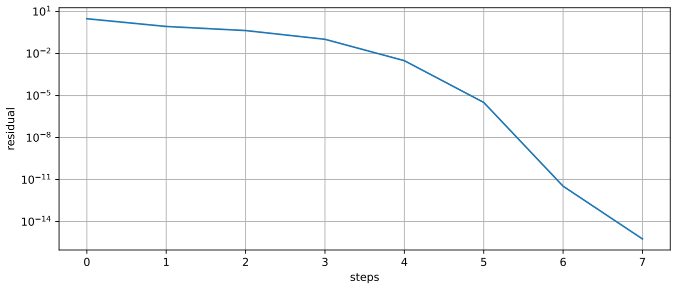
```
A1(u)=M(u)*u
```

newton1 (generic function with 1 method)

```
function newton1(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, 3.3511

```
newton_result1,newton_history1=newton1(A1,F,[0,0.1],tol=1.e-13)
```




```
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)

```
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]
```

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]
```

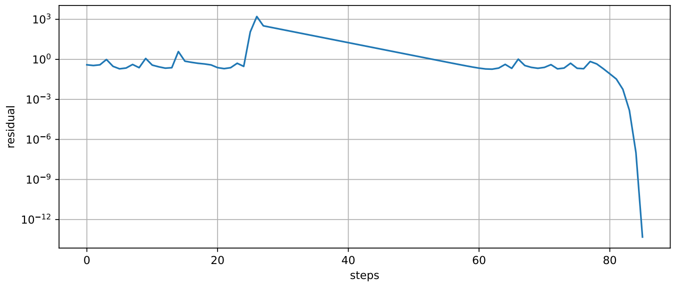
([-0.188484, 0.198519, 0.488388], [0.39077, 0.345694, 0.389908, 0.977557, 0.300465, 0.1952

```
res2,hist2=newton1(A2,F2,U02)
```

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

```
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.

dnewton: 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)
```

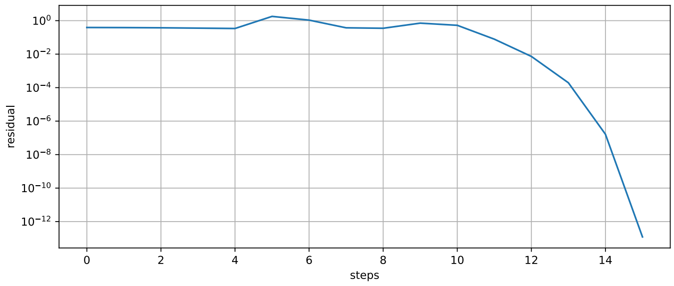
```
function dnewton(A,b,u0; tol=1.0e-12,maxit=100,damp=1,damp_growth=1)
    result=DiffResults.JacobianResult(u0)
    history=Float64[]
    u=copy(u0)
    it=1
    while it<maxit
        ForwardDiff.jacobian!(result,(v)->A(v)-b ,u)
        res=DiffResults.value(result)
        jac=DiffResults.jacobian(result)
        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, 1.79877
```

```
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)

```
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=newton1(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
```

```
(
  1:   [-0.188484, 0.198519, 0.488388]
  2:   [[100.408, 1.41554e-14], [28.0258, 16.6762, 13.3379, 10.6677, 8.53262,   more ,3.
)
```

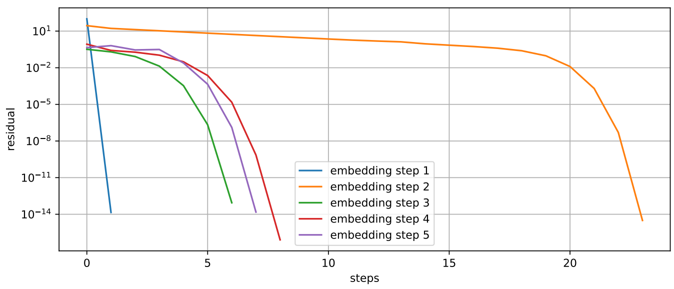
```
res4,hist4=embed_newton(A2λ,F2,U02,δ0=0.01,δgrowth=5.0)
```

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

```
A2λ(res4,1.0)-F2
```

Newton steps: 50

plothistory (generic function with 2 methods)




```
plothistory(hist4)
```

NLsolve.jl

```
using NLsolve
```

```
WARNING: method definition for TwiceDifferentiable at /home/fuhrmann/.julia/
packages/NLSolversBase/cfJrN/src/objective_types/incomplete.jl:96 declares type
variable TH but does not use it.
```

```
nlres1 = Results of Nonlinear Solver Algorithm
          * Algorithm: Trust-region with dogleg and autoscaling
          * Starting Point: [1.0, 1.0, 1.0]
          * Zero: [0.057582447577986924, 0.4839954302915904, 0.04126490295783218]
          * 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): 83
          * Jacobian Calls (df/dx): 40
```

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

```
[0.0175049, -2.60128e-5, -0.0880858]
```

```
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: [-0.18848435786947373, 0.198519144942218, 0.4883882611017444]
          * Inf-norm of residuals: 0.000000
          * Iterations: 239
          * Convergence: true
            * |x - x'| < 0.0e+00: false
            * |f(x)| < 1.0e-08: true
          * Function Calls (f): 240
          * Jacobian Calls (df/dx): 240
```

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

```
[-1.12965e-14, 8.32667e-17, 7.83734e-13]
```

```
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: [-0.18848435786937287, 0.19851914494226677, 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)
```

```
[-7.91034e-15, 5.27356e-16, 1.06304e-13]
```

```
A2λ(nlres3.zero,1.0)-F2
```

Summary

- 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