```
true
```

```julia
begin
    ENV["LANG"]="C"
    using Pkg
    Pkg.activate(mktempdir())
    Pkg.add(["PyPlot","PlutoUI","DualNumbers","ForwardDiff","DiffResults"])
    using PlutoUI
    using PyPlot
    using DualNumbers
    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 ot i with $i^2 = -1$.

Dual numbers are defined by extending the real numbers by formally adding an number ε with $\varepsilon^2 = 0$:

$$D = \{a + b\varepsilon \mid a, b \in \mathbb{R}\} = \left\{ \begin{pmatrix} a & b \\ 0 & a \end{pmatrix} \mid a, b \in \mathbb{R} \right\} \subset \mathbb{R}^{2\times 2}$$

They 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

Constructing a dual number:

```
d = 2 + 1ε
d=Dual(2,1)
```

Accessing its components:

```
(2,  1)
d.value,d.epsilon
```

Comparison with known derivative:

```
testdual (generic function with 1 method)
function testdual(x,f,df)
    xdual=Dual(x,1)
    fdual=f(xdual)'
    (f=f(x),f_dual=fdual.value),(df=df(x),df_dual=fdual.epsilon)
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 = 34.0,  f_dual = 34.0),  (df = 29.0,  df_dual = 29.0))
testdual(3.0,p,dp)
```

Standard functions:

```
((f = -0.544021,  f_dual = -0.544021),  (df = -0.839072,  df_dual = -0.839072))
testdual(10,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))
```

Conclusion: 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.

The forwardiff package provides 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
```

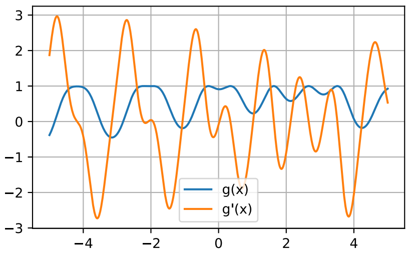
```
(f = 0.420167,  df = 0.907447,  df_dual = 0.907447)
testdual1(13,sin,cos)
```

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, tol)
    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)
    [ 0.1+(u[1]^2+u[2]^2)  -(u[1]^2+u[2]^2);
        -(u[1]^2+u[2]^2)  0.1+(u[1]^2+u[2]^2)]
end
```

```
F =  Int64[1,  3]
```

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

```
(Float64[19.9994,  20.0006],  Float64[3.16228,  28284.3,  0.282829,  4.95196e-10,  1.81899e
```

```
fixpt_result,fixpt_history=fixpoint!([0,0],M,F,100,1.0e-12)
```

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

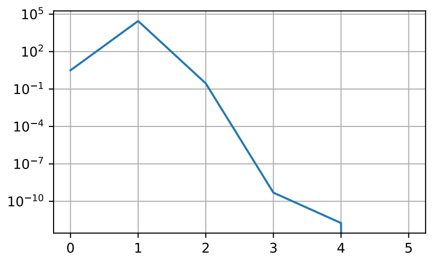
```
contraction(h)=h[2:end]./h[1:end-1]
```

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

```
function plothistory(history)
    clf()
    semilogy(history)
    grid()
    gcf()
end
```

```
Float64[8944.27,  9.9995e-6,  1.75087e-9,  0.00367327,  0.0]
```

```
contraction(fixpt_history)
```



```
plothistory(fixpt_history)
```

```
Float64[0.0,  0.0]
```

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

Newton iteration scheme

The fixed point iteration scheme assumes a particular structure of the nonlinear system. 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

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

The 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 folows:

- 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: Asymptically 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([5.0e-324, 5.0e-324], ([6.91366206214077e-310 6.91366204885713e-310; 6.9
```

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

Calculate function and derivative at once:

```
MutableDiffResult([0.1999999999999993, 0.1999999999999993], ([8.100000000000001 -8.0; -8.0
```

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

```
Float64[0.2,  0.2]
```

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

```
2×2 Array{Float64,2}:
  8.1  -8.0
 -8.0   8.1
```

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

A Newton solver with automatic differentiation

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

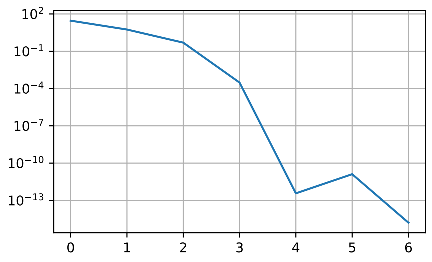
```
function newton(A,b,u0; tol=1.0e-12, maxit=100)
    result=DiffResults.JacobianResult(u0)
    history=Float64[]
    u=copy(u0)
    it=1
    while it<maxit
        ForwardDiff.jacobian!(result,(v)->A(v)-b ,u)
        res=DiffResults.value(result)
        jac=DiffResults.jacobian(result)
        h=jac\res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

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

```
(Float64[19.9994, 20.0006], Float64[28.8467, 5.58664, 0.493295, 0.000301159, 3.69765e
```

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

```
Float64[0.193667, 0.0882991, 0.000610505, 1.22781e-9, 34.7848, 0.000124992]
```

```
contraction(newton_history)
```



```
plothistory(newton_history)
```

```
Float64[1.81899e-12, -1.81899e-12]
```

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

Let us take a more complicated example:

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

```
A2(x)= [x[1]+x[1]^5+3*x[2]*x[3],
        0.1*x[2]+x[2]^5-3*x[1]-x[3],
        x[3]^5+x[1]*x[2]*x[3]]
```

```
F2 =  Float64[0.1, 0.1, 0.1]
```

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

```
U02 =  Float64[1.0, 1.0, 1.0]
```

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

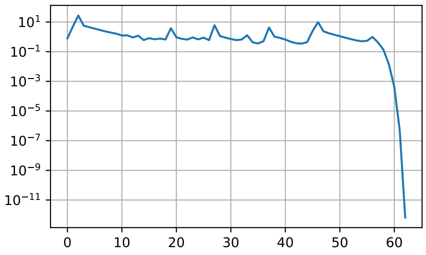
```
(Float64[-0.248731, 0.175566, 0.663915], Float64[0.796625, 4.90091, 27.5487, 5.62444,
res2,hist2=newton(A2,F2,U02)
```

```
Float64[0.0,  -1.38778e-16,  -1.38778e-17]
```

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

```
(63,  Float64[6.15208,  5.62115,  0.204163,  0.799647,  0.80018,  0.800945,  0.803928,  0.8
```

```
length(hist2),contraction(hist2)
```



```
plothistory(hist2)
```

Here, we observe that we have to use lots of iteration steps and see a rather erratic behaviour of the residual. After ≈ 55 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
- linesearch: automatic detection of a damping factor

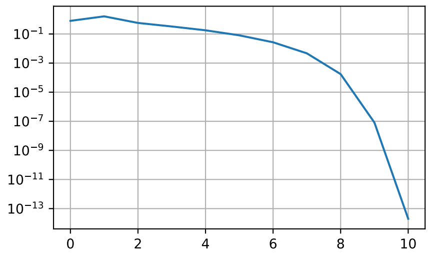
```
dnewton (generic function with 1 method)
```

```
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
```

```
(Float64[-0.248731,  0.175566,  0.663915],  Float64[0.796625,  1.62137,  0.572359,  0.32642
```

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

```
(11,  Float64[2.0353,  0.35301,  0.570316,  0.546644,  0.45155,  0.332823,  0.174192,  0.03
length(hist3),contraction(hist3)
```



```
plothistory(hist3)
```

```
Float64[-2.77556e-17,  -2.77556e-17,  -1.38778e-17]
```

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

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

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
- Parameter embedding + damping + update based convergence control go a long way to solve even strongly nonlinear problems!
- As we will see later, a similar apporach can be used for time dependent problems.