

## Advanced Topics from Scientific Computing

TU Berlin Winter 2022/23

## Notebook 09

 Jürgen Fuhrmann

```

- begin
-   using LinearAlgebra
-   using ForwardDiff, DiffResults, NLsolve, RecursiveArrayTools
-   using RungeKutta
-   using DifferentialEquations
-   using DiffEqDevTools
- end

```

## ODE and DAE Solvers

Literature:

- E. Hairer, S.P. Nørsett, G. Wanner: Solving Ordinary Differential Equations I. Nonstiff Problems
- E. Hairer, G. Wanner: Solving Ordinary Differential Equations II. Stiff and Differential Algebraic Problems
- P. Kunkel, V. Mehrmann: Differential-Algebraic Equations. Analysis and Numerical Solution

Further sources used:

- [TU Chemnitz lecture notes by A. Naumann](#) (in German)
- HU Berlin course "Numerik gewöhnlicher Differentialgleichungen I" (implementation of Explicit Runge-Kutta)

While this course focuses on the solution of partial differential equations which describe coupled processes in space and time, we first talk about the solution of ordinary differential equations, providing methods for discretization in time.

Look for  $\mathbf{u} : [0, T] \rightarrow \mathbb{R}^m$  be a time dependent, differentiable function such that it fulfills the initial value problem for the ordinary differential equation (ODE) system:

$$\begin{cases} \dot{\mathbf{u}} &= \mathbf{f}(\mathbf{u}, \mathbf{p}, t) \\ \mathbf{u}|_{t=0} &= \mathbf{u}_0 \end{cases}$$

with  $\mathbf{f} : \mathbb{R}^m \times \mathbb{R}^k \times [0, T] \rightarrow \mathbb{R}^m$ .

Here,  $\mathbf{p}$  is a  $k$ -vector of parameters.

The system is called *autonomous* if  $\mathbf{f}$  does not depend on  $t$ .

Subdivide  $[0, T]$  into  $N - 1$  intervals of time step size  $\tau = \frac{T}{N-1}$ . Let  $t_n = (n - 1)\tau$  and let  $\mathbf{u}_n = \mathbf{u}(t_n)$ .

## Explicit Euler method

The simplest way of discretizing in time is to equate the finite difference in time to the right hand side value calculated in the last timestep.

$$\frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{\tau} = \mathbf{f}(\mathbf{u}_n, \mathbf{p}, t_n)$$

We implement this *explicit Euler method* such that we describe the right hand side  $\text{du}=\mathbf{f}(\mathbf{u}, \mathbf{p}, t)$  via a mutating function  $\mathbf{f}!(\text{du}, \mathbf{u}, \mathbf{p}, t)$ . It returns a vector of solution vectors and the chosen time values.

```

- function explicit_euler(f!::F, # ODE right hand side
-   p; # Parameters
-   u0=[0.0], # vector of initial values
-   dt=0.1, # time step size
-   tspan=[0.0, 1.0] # time interval
-   ) where F
-   u=[u0]
-   t=Float64[tspan[1]]
-   du=zeros(length(u0))
-   while t[end]<tspan[end]
-     uold=u[end]
-     told=t[end]
-     f!(du, uold, p, told)
-     unew=uold+dt*du
-     tnew=toold+dt
-     push!(u, unew)
-     push!(t, tnew)
-   end
-   u, t
- end;

```

## Test problem

Let us first look at the equation

$$\dot{\mathbf{u}} = \lambda \mathbf{u}$$

```

- f_test!(du, u, λ, t) = du[1]=λ*u[1];

```

The exact solution is given by  $u = u_0 e^{\lambda t}$ .

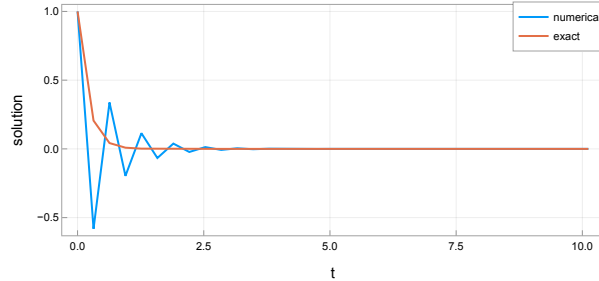
```
· exact_test(u0,p,t)=u0*exp(p*t);
```

For  $\lambda = \frac{\log \frac{1}{2}}{t_{\frac{1}{2}}}$ , this equation e.g. describes radioactive decay with half value time  $t_{\frac{1}{2}}$

```
p_test = -5
· p_test=-5
```

$\log(\tau) =$   -0.5

```
· u_test,t_test= explicit_euler(f_test!,p_test; u0=[1.0],dt=10*logdt_test,tspan=(0,10));
```



Obviously we may have a problem with the step size, and we may try to find other methods.

## θ- and Runge-Kutta methods

Main theorem of calculus gives

$$\int_{t_n}^{t_{n+1}} \dot{u} = \int_{t_n}^{t_{n+1}} f(u(t), p, t)$$

Approximate the integral using a quadrature rule with points  $c_i$  and weights  $b_i$ :

$$u_{n+1} - u_n = \tau \sum_{i=1}^s b_i f(u_{n,i}, p, t_n + c_i \tau)$$

Using some simple quadratures leads to

$$b = \{1\}, c = \{\theta\}, u_{n,i} = \theta u_{n+1} + (1 - \theta) u_n$$

- $\theta = 0$ : explicit Euler method
- $\theta = 1$ : implicit Euler method
- $\theta = \frac{1}{2}$ : implicit midpoint method

$$\frac{u_{n+1} - u_n}{\tau} = f(\theta u_{n+1} + (1 - \theta) u_n, p, t_n + \theta \tau)$$

Unless  $\theta=0$  (explicit Euler method) we need to solve an equation or a system of equations in each timestep. Here we implement this for general  $\theta$ .

### θ method: implementation

This implementation is slightly more general:

- if `linear=true`, instead of Newton's method to solve the implicit equations it performs just one Newton step (assuming the time step size is small enough). This results in a *linear implicit* method (linear implicit Euler or linear implicit midpoint)
- it implements the solution of a slightly more general problem: for a matrix M (unit matrix by default), solve

$$M \dot{u} = f(u, p, t).$$

```

function Omethod(f!::F, # right hand side function
                p; # Parameters
                u0=[0.0], # Initial value
                dt=0.1, # time step size
                tspan = (0,1), # time interval
                θ=0, # θ parameter
                linear=false, # switch between linear method and newton solver
                M=nothing, # Mass matrix
                tol=1.0e-10, # Newton method tolerance
                ) where F
    # Start arrays for the result
    u=[u0]
    t=Float64[tspan[1]]
    uold=u[end]
    told=t[end]

    # If no mass matrix is given as parameter, let it be the unit matrix of
    # appropriate size
    if M==nothing
        M=Diagonal(ones(length(u0)))
    end

    # Each timestep involves the solution of F(uneu)=0
    # Here, uneu is the current iteration and F is the result
    function fstep!(F,uneu)
        f!(F, θ*uneu+(1-θ)*uold, p, told+θ*dt)
        F.=M*(uneu .- uold)./dt .- F
    end

    if linear
        # Handle differentiation results
        diffresult=DiffResults.JacobianResult(u0)
        y=zero(u0)
        cfg = ForwardDiff.JacobianConfig(fstep!,y, u0)
    end

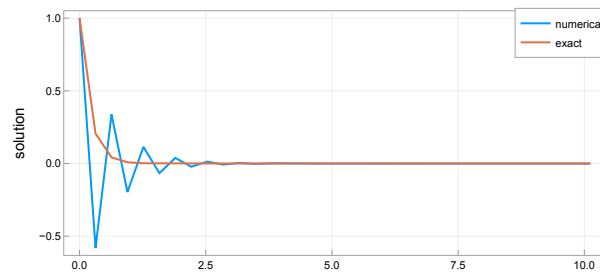
    while t[end]<tspan[2]
        # Loop over the timesteps
        told=t[end]
        uold=u[end]
        if linear
            # Perform one Newton step
            ForwardDiff.jacobian!(diffresult, fstep!,y,uold,cfg)
            uneu=uold-DiffResults.jacobian(diffresult)\DiffResults.value(diffresult)
        else
            # Solve time step problem and check for convergence
            result=nl_solve(fstep!,uold; autodiff=:forward,xtol=tol)
            !converged(result) && throw("convergence error")
            uneu=result.zero
        end
        # Extend solution arrays
        push!(u,uneu)
        push!(t,told+dt)
    end
    # Return the solution as DiffEqArray
    # from RecursiveArrayTools.jl (easy to plot)
    DiffEqArray(u,t)
end;

```

So let us use this for our test example:

```
u0_test=Omethod(f_test!,p_test; u0=[1.0],dt=10^logdtx,tspan=(0,10),θ=θx);
```

θ =  0.0 log(τ) =  -0.5



## Runge-Kutta methods

A generalization of the the choice of internal stages  $u_{n,i}$  in the quadrature based method starting with

$$\int_{t_n}^{t_{n+1}} \dot{u} = \int_{t_n}^{t_{n+1}} f(u(t), p, t)$$

$$u_{n+1} - u_n = \tau \sum_{i=1}^s b_i f(u_{n,i}, p, t_n + c_i \tau)$$

chooses another quadrature rule with the points  $c_1 \dots c_s$  and weights  $a_{ij}$ :

$$u_{n,i} - u_n = \tau \sum_{j=1}^s a_{ij} f(u_{n,j}, p, t_n + c_j \tau)$$

This Ansatz gives s-stage Runge-Kutta methods.

If A is strictly lower triangular, the method is explicit, i.e. it does only involve the evaluation of  $f$ , no solution of a system of equations involving  $f$ .

Otherwise, it is implicit, and it involves the solution of one or more, possibly coupled systems of equations of dimension  $m \times s$ . As a consequence, implicit Runge-Kutta methods are hard to use for large systems.

Runge-Kutta methods thus are described by the points  $c = \{c_1 \dots c_s\}$ , the weights  $b = \{b_1 \dots b_s\}$  and the  $s \times s$  matrix  $A = (a_{ij})$ . Usually these are arranged into *Butcher tableaux*:

c	A
	b

These tableaux can be conveniently managed using the `RungeKutta.jl` package.

The  $\theta$  method is a Runge-Kutta method:

Runge-Kutta Tableau `explicit_euler` with 1 stages and order 1:

0//1	0//1
	1//1

· `TableauExplicitEuler(Rational)`

Runge-Kutta Tableau `implicit_euler` with 1 stages and order 1:

1//1	1//1
	1//1

· `TableauImplicitEuler(Rational)`

Runge-Kutta Tableau `implicit_midpoint` with 1 stages and order 2:

1//2	1//2
	1//1

· `TableauImplicitMidpoint(Rational)`

The classical Runge-Kutta method is "RK4":

Runge-Kutta Tableau `erk4` with 4 stages and order 4:

0.0	0.0	0.0	0.0	0.0
0.5	0.5	0.0	0.0	0.0
0.5	0.0	0.5	0.0	0.0
1.0	0.0	0.0	1.0	0.0
	0.166667	0.333333	0.333333	0.166667

· `TableauRK4(Float64)`

With `RungeKutta.jl`, one can e.g. define the Dormand-Prince tableau:

Runge-Kutta Tableau `DoPri5` with 7 stages and order 5:

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.2	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.3	0.075	0.225	0.0	0.0	0.0	0.0	0.0	0.0
0.8	0.977778	-3.733333	3.555556	0.0	0.0	0.0	0.0	0.0
0.888889	2.9526	-11.5958	9.82289	-0.290809	0.0	0.0	0.0	0.0
1.0	2.84628	-10.7576	8.90642	0.278409	-0.273531	0.0	0.0	0.0
1.0	0.0911458	0.0	0.449236	0.651042	-0.322376	0.130952	0.0	0.0
	0.0911458	0.0	0.449236	0.651042	-0.322376	0.130952	0.0	0.0

1 column omitted

· `TableauDP(Float64,5)`

`TableauDP` (generic function with 4 methods)

The Radau IIA method is an implicit Runge-Kutta method:

Runge-Kutta Tableau `RadauIIA(3)` with 3 stages and order 5:

0.155051	0.196815	-0.0655354	0.023771
0.644949	0.394424	0.292073	-0.0415488
1.0	0.376403	0.512486	0.111111
	0.376403	0.512486	0.111111

· `TableauRadauIIA(3)`

Implementation of the explicit Runge-Kutta scheme:

```

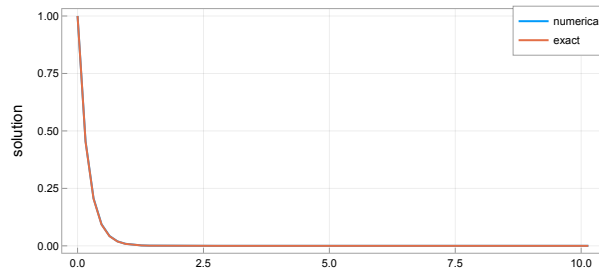
explicit_rk (generic function with 1 method)
- function explicit_rk(f!::F, # ODE right hand side
-   p; # Parameters
-   tableau=TableauRK4(),
-   u0=[0.0], # vector of initial values
-   dt=0.1, # time step size
-   tspan=(0,1), # time interval
-   ) where F
-
-   u=[u0]
-   t=Float64[tspan[1]]
-   a=tableau.a
-   b=tableau.b
-   c=tableau.c
-   m=length(u0)
-   s=length(b)
-   du=zeros(m)
-   z=zeros(m,s)
-   while t[end]<tspan[end]
-     z.=0
-     for j = 1 : s
-       du.=0
-       for k = 1 : j-1
-         @views du .+= z[:,k] * a[j,k]
-       end
-       @views f!( z[:,j], u[end] .+ dt * du, p, t[end] + c[j]*dt)
-     end
-     du.=0
-     for j = 1 : s
-       @views du.+= b[j] * z[:,j]
-     end
-     u_new=u[end]+dt*du
-     t_new=t[end]+dt
-     push!(u,u_new)
-     push!(t,t_new)
-   end
-   DiffEqArray(u,t)
- end

- tableau=TableauRK4();

- urk_test= explicit_rk(f_test!,p_test; u0=[1.0], tableau, dt=10*logdt_rk,tspan=(0,10));

```

$\log(\tau) =$   -0.8



Once again we see that for a "decent" solution, the time step size should be not too large. We need to discuss two phenomena:

- What about the "wiggles" which destroy the overall quality of solutions  $\Rightarrow$  stability
- How good is the approximation  $\Rightarrow$  approximation order

## Stability

Let us discuss aspects of stability (resp. instability).

Accidentally, the test problem  $\dot{\mathbf{u}} = \lambda \mathbf{u}$  chosen here is also called *Dahlquist test equation*. It is used to judge the stability of ODE methods.

### A-Stability

**Definition:** An ODE method is called unconditionally A-stable if for  $\lambda \in \mathbf{C}$  with  $\text{Re}\lambda < 0$ , the approximate solution of the Dahlquist test equation fulfills  $|\mathbf{u}_{n+1}| \leq |\mathbf{u}_n|$ .

The method is called conditionally A stable if  $|\mathbf{u}_{n+1}| \leq |\mathbf{u}_n|$  is fulfilled for  $h\lambda \in \mathcal{S} \subset \mathbf{C}$ , where  $\mathcal{S}$  is called stability region.

In order to investigate A-Stability, it is useful to define the stability function R such that

$$\mathbf{u}_{n+1} = \mathbf{R}(\tau\lambda)(\mathbf{u}_n)$$

For the  $\theta$ -methods this gives:

$$\mathbf{R}_0(z) = 1+z;$$

$$\mathbf{R}_{05}(z) = (1+z/2)/(1-z/2);$$

$$\mathbf{R}_1(z) = 1/(1-z);$$

For generic Runge-Kutta methods, we have:

```
function R_rk(z,tableau=TableauRK4())
    A=tableau.a
    b=tableau.b
    Ib=ones(length(b))
    1 + z*b' * inv(I-z*A)*Ib
end;
```

### L-Stability

An A-stable one-step method is called L-stable if for all  $\lambda \in \mathbb{C}$  with  $\text{Re}\lambda < 0$ , the solution of the Dahlquist test problem fulfils

$$\lim_{n \rightarrow \infty} |u_n| = 0$$

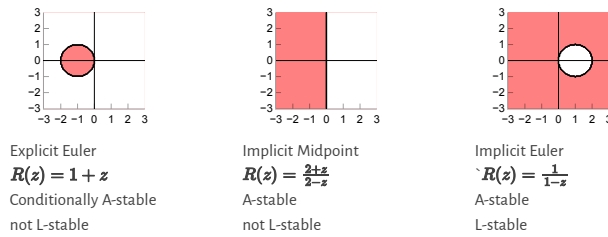
independent of the stepsize  $\tau$ .

L-Stability is fulfilled if and only if for all  $\lambda \in \mathbb{C}$  with  $\text{Re}\lambda < 0$ ,

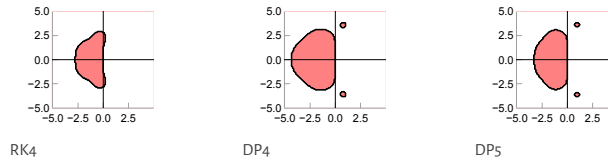
$$\lim_{\tau \rightarrow \infty} R(\tau\lambda) = 0$$

stabregion (generic function with 1 method)

Stability of the theta methods:



Explicit Runge-Kutta methods are conditionally A stable:



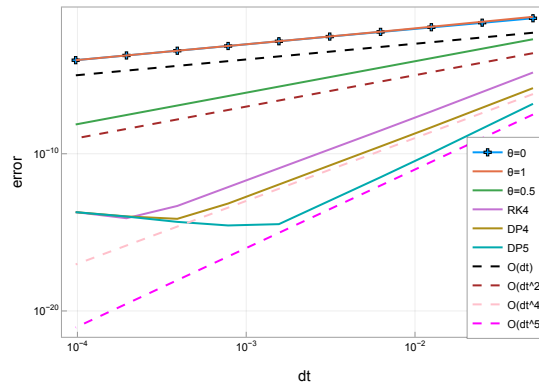
Implicit Runge-Kutta methods are A-stable, some of them are L-stable like RadauIA(5):



### Approximation order

Approximation order can generally be analysed using Taylor expansion to estimate the consistency order. We only demonstrate the results here.

errortest (generic function with 3 methods)



- Implicit Euler and explicit Euler exhibit first order of approximation
- Implicit midpoint has second approximation order
- Runge-Kutta methods can have higher order

These methods are one-step methods, which just involve two subsequent time steps. Multistep methods (e.g. BDF - backward differencing formula) achieve higher order by taking into account more of the older time steps, however they have a problem to keep their order at the start.

## An ODE system

We demonstrate the Lotka-Volterra system describing predator-prey dynamics:

- species x: prey which growth rate  $a$  being eaten with rate  $b$  by the predators
- species y: predators growing with rate  $d$  by eating prey and dying with rate  $c$

lotkavolterra! (generic function with 1 method)

```
function lotkavolterra!(du,u,p,t)
    (a,b,c,d)=p
    x,y=u[1],u[2]
    du[1]=a*x - b*x*y
    du[2]=-c*y + d*x*y
end
```

The function V is constant along trajectories – it is easy to show that  $\frac{dV}{dt} = 0$ . Therefore in the phase-space the trajectories are isolines of  $V(x, y)$ , and the solution must be periodic.

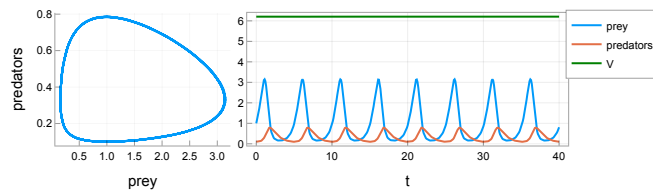
```
function V(x::Number,y::Number,param)
    a,b,c,d=param
    x=max(1.0e-15,x)
    y=max(1.0e-15,y)
    d*x - c*log(x) + b*y - a*log(y)
end;
```

param = (a = 2, b = 6, c = 1, d = 1)

```
param=(a=2,b=6,c=1,d=1)
```

```
u1v=let
    rk1v ? explicit_rk(lotkavolterra!,param; u0=[1.0,0.1],dt=10*logdtlv,tspan=(0,40)) :
    @method(lotkavolterra!,param; u0=[1.0,0.1],dt=10*logdtlv,tspan=(0,40),θ=θlv,linear=linlv)
end;
```

θ =  log(dt) =  Linear =  RK4 =



- The implicit midpoint rule catches periodicity - surprisingly also for rather coarse time steps. The reason is that it preserves more of the structure of the system. This is the simplest *geometrical integrator* - a class of methods which exactly preserves certain invariants of the solution
- The Runge-Kutta methods does a quite good job in catching periodicity
- Implicit and explicit Euler are unable to catch the dynamics in the right way.
- The linear implicit approach works well

## Stiff problems

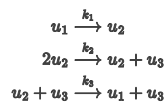
Problems with a high ratio between the real parts of the largest and the smallest eigenvalues of  $\frac{\partial f}{\partial u}$  (spectral condition number  $\kappa$  for spd problems) are called *stiff*.

This is e.g. the case with different time scales which may be due to fast and slow reactions, and for systems arising from the discretization of PDEs, where the condition number increases with  $O(h^{-2})$ , where  $h$  is the space discretization parameter.

Stability then depends on the stability constraint for the smallest timescale.

Unconditionally A-stable and L-stable methods are to be preferred if we want to guarantee stability for stiff problems.

Let us consider a chemical reaction ("Robertson reaction" example from matlab)



fstiff! (generic function with 1 method)

```
function fstiff!(du,u,p,t)
    k1,k2,k3=p
    du[1]= -k1*u[1] + k3* u[2]*u[3]
    du[2]= k1*u[1] - k2* u[2]*u[3] - k3*u[2]^2
    du[3]= k2*u[2]^2
end
```

For this problem,  $d$  can be seen as "stiffness" parameter. The larger  $d$  the shorter the time scale of the second equation.

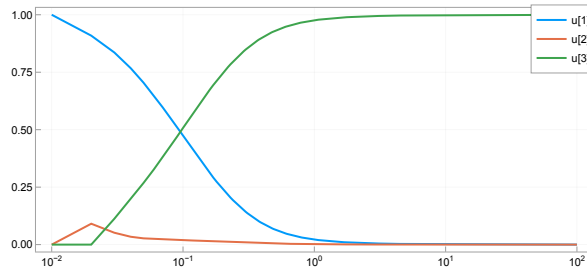
```
probertson = (k1 = 10, k2 = 10000.0, k3 = 100)
```

```
· probertson=(k1=10,k2=10^4loga,k3=100)
```

```
· ustiff=0method(fstiff!,probertson; u0=[1.0,0.0, 0.0],dt=10^logdtx2,θ=θx2,
  linear=lin2, tspan=(1.0e-2,100));
```

θ =  1.0 log(dt) =  -2.0 linear =

Stiffness log(a) =  4.0



Due to the fact that the implicit midpoint rule is not L-stable, we can have problems with large stiffness.

## DAE problems

Differential-algebraic equation (DAE) system:

$$0 = f(\dot{u}, u, p, t)$$

Here, we will focus on mass matrix DAE systems (MMDAE):

$$M\dot{u} = f(u, p, t)$$

M is a diagonal matrix, some entries may be zero. The corresponding equations in the system turn from differential equations to algebraic equations.

We can see a DAE as the "large stiffness limit" case of an ODE system, so once again we want to use A-stable and L-stable methods.

Here, we re-define the Robertson example using the fact that the sum of all three reactants must be 1.

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

```
· function fdae!(du,u,p,t)
  · k1,k2,k3=p
  · du[1]= -k1*u[1] + k3*u[2]*u[3]
  · du[2]= k1*u[1] - k3*u[2]*u[3] - k2*u[2]^2
  · du[3]= u[1] + u[2] + u[3] - 1
  · end
```

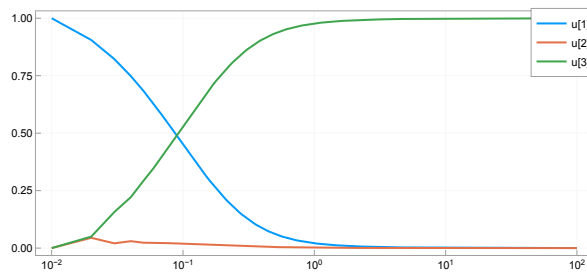
```
M_DAE = 3x3 Diagonal{Float64, Vector{Float64}}:
```

```
 1.0 . .
 . 1.0 .
 . . 0.0
```

```
· M_DAE=Diagonal([1.0,1,0])
```

θ =  0.5 log(dt) =  -2.0 linear =

Stiffness log(a) =  4.0



```
· plot(udae, label=["u[1]" "u[2]" "u[3]"],size=
  (600,300),linewidth=2,framestyle=:box,xscale=:log10)
```

As for stiff problems, we want implicit methods to solve DAE problems.

## Multistep methods

Multistep methods attempt to take into account information from timesteps prior to  $t_n$ .

### BDF Methods



For a given vector  $\mathbf{X} = \mathbf{x}_1 \dots \mathbf{x}_n$ , and function values  $\mathbf{Y} = \{y_i = f(\mathbf{x}_i)\}_{i=1..n}$  a Lagrange polynomial  $L[\mathbf{X}, \mathbf{Y}]$  interpolates the function  $f$  in the interval  $[\mathbf{x}_1, \mathbf{x}_n]$ . It is given by

$$L[\mathbf{X}, \mathbf{Y}](\xi) = \sum_{j=1}^n l_j[\mathbf{X}](\xi) y_j$$

with

$$l_j[\mathbf{X}](\xi) = \prod_{i=1..n, i \neq j} \frac{\xi - \mathbf{x}_i}{\mathbf{x}_j - \mathbf{x}_i}$$

From the very definition we have  $l_j[\mathbf{X}](\mathbf{x}_i) = \delta_{ij}$ . The interesting point is that we can hope that  $L'[\mathbf{X}](\mathbf{x}_n)$  is a good approximation of  $f'(\mathbf{x}_n)$ . This idea gives another way to approximate the derivative on the left hand side of the ODE, taking into account results from older timesteps.

In particular, for the case of three intrpolation points  $\mathbf{X} = \{a, b, c\}$  we get

$$\begin{aligned} l_a[\mathbf{X}](\xi) &= \frac{(\xi - b)(\xi - c)}{(a - b)(a - c)} & l'_a[\mathbf{X}](\xi) &= \frac{2\xi - b - c}{(a - b)(a - c)} & l'_a[\mathbf{X}](c) &= \frac{c - b}{(a - b)(a - c)} \\ l_b[\mathbf{X}](\xi) &= \frac{(\xi - a)(\xi - c)}{(b - a)(b - c)} & l'_b[\mathbf{X}](\xi) &= \frac{2\xi - a - c}{(b - a)(b - c)} & l'_b[\mathbf{X}](c) &= \frac{c - a}{(b - a)(b - c)} \\ l_c[\mathbf{X}](\xi) &= \frac{(\xi - a)(\xi - b)}{(c - a)(c - b)} & l'_c[\mathbf{X}](\xi) &= \frac{2\xi - a - b}{(c - a)(c - b)} & l'_c[\mathbf{X}](c) &= \frac{2c - a - b}{(c - a)(c - b)} \end{aligned}$$

Let  $a = t^{n-1}, b = t^n, c = t^{n+1}$

Setting  $\mathbf{Y} = \{y_a, y_b, y_c\}$  gives

$$L'[\mathbf{X}, \mathbf{Y}](c) = \frac{1}{2\tau} (y_a - 4y_b + 3y_c)$$

For  $y_c = u^{n+1}, y_b = u^n, y_a = u^{n-1}$  gives rise to the BDF2 (Backward Differencing Formula) scheme:

$$\frac{1}{2} (3u^{n+1} - 4u^n + u^{n-1}) = \tau f(u^{n+1}, p, t^{n+1})$$

BDF schemes are implicit. Unlike the higher order pendants, BDF1 and BDF2 are A-stable, and thus useful for stiff problems. BDF2 has consistency order 2.

It is easy to see that this procedure can be adapted to varying step sizes. Further refined variants ensure that under stepsize variations the coefficient before  $u_{n+1}$  stays unchanged ([Fixed leading coefficient BDF in SUNDIALS](#); [FBDF in Julia](#)). More information is available via [Scholarpedia](#).

A higher order correction of the BDF methods leads to the QNDF method (quasi constant timestep numerical differentiation formula) which is equivalent to the `ode15s` method implemented in the [matlab ODE suite](#). QNDF is the [preferred default solver for stiff systems in Julia](#)

### Adams methods

A similar interpolation approach can be applied to the right hand side of the ODE system, giving rise e.g. to the explicit Adams-Bashforth methods.

## DifferentialEquations.jl

So far, we did our own implementations of some methods discussed These implementations miss a number of points:

- Time step adaptivity. A general approach to time step adaptivity considers the comparison between two solutions of different consistency order to obtain an error estimate.
- Multistage linear implicit methods (Rosenbrock methods). The simple methods of this class are the linear implicit methods which replace the nonlinear solution by just one Newton step. Higher order variants combine a fixed number of intermediate linear steps.

Julia provides a state of the art toolbox for the solution of systems of differential equations:

[DifferentialEquations.jl](#) The overview on implemented methods is given in the [documentation](#):

This documentation is not for the latest stable release, but for either the development version or an older release.  
[Click here to go to the documentation for the latest stable release.](#)

```
alg = Tsit5()
solve(prob,alg)
```

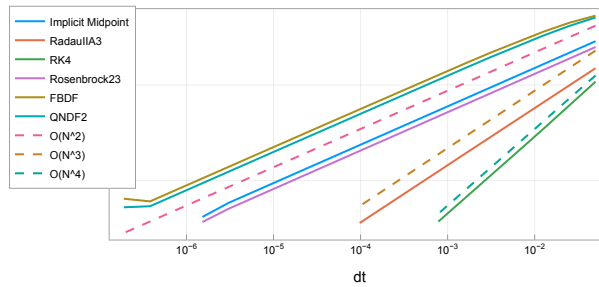
Additionally, the following algorithms have a lazy interpolant:

- BS5 - Bogacki-Shampine 5/4 Runge-Kutta method. (lazy 5th order interpolant).
- Vern6 - Verner's "Most Efficient" 6/5 Runge-Kutta method. (lazy 6th order interpolant).
- Vern7 - Verner's "Most Efficient" 7/6 Runge-Kutta method. (lazy 7th order interpolant).
- Vern8 - Verner's "Most Efficient" 8/7 Runge-Kutta method. (lazy 8th order interpolant).
- Vern9 - Verner's "Most Efficient" 9/8 Runge-Kutta method. (lazy 9th order interpolant).

These methods require a few extra steps in order to compute the high order interpolation, but these steps are only taken when the interpolation is used. These methods when lazy assume that the parameter vector  $p$  will be unchanged between the moment of the interval solving and the interpolation. If  $p$  is changed in a ContinuousCallback, or in a DiscreteCallback and the continuous solution is used after the full solution, then set `lazy=false`.

We choose a subset of the methods implemented there and return to the example problems we discussed:

```
diffeqmethods = OrderedCollections.OrderedDict{String, Type}()
    "Implicit Euler" => ImplicitEuler
    "Implicit Midpoint" => ImplicitMidpoint
    "RadauIIA3 (Implicit RK)" => RadauIIA3
    "RK4 (Explicit RK)" => RK4
    "Rosenbrock23 (Rosenbrock)" => Rosenbrock23
    "QNDF2" => QNDF2
    "FBDF" => FBDF
)
```



### Lotka-Volterra revisited

analyze (generic function with 1 method)

```
param2 = (a = 2, b = 6, c = 1, d = 1)
param2=(a=2,b=6,c=1,d=1)
```

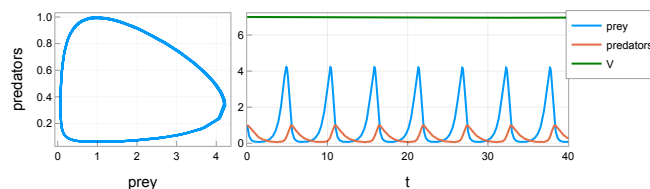
```
ulv_deq=let
    prob=ODEProblem(lotkaVolterra!, [1.0, 1.0], (0, 40), param2)
    method=diffeqmethods[lvmethod]
    if method==ImplicitMidpoint
        dt = 10*logreltol
        @info "implicit midpoint: dt=$(dt)"
        solve(prob, method(); dt);
    else
        reltol=10*logreltol
        @info "reltol=$(reltol)"
        solve(prob, method(); reltol);
    end
end;
```

reltol=0.001

method:  log10(reltol)/log10(dt):

(timesteps = 475, smallest = 8.5835e-7, largest = 0.21573)

```
analyze(ulv_deq)
```



## Stiffness + DAE revisited

```

- ustiff_deq=let
-   prob=ODEProblem(fstiff!, [1.0, 0, 0], (1.0e-2, 10), (k1=10, k2=10*logst, k3=100))
-   solve(prob, diffeqmethods[stmethod](); reltol=10*logstreltol);
- end;

```

(timesteps = 49, smallest = 9.999e-6, largest = 2.08807)

```

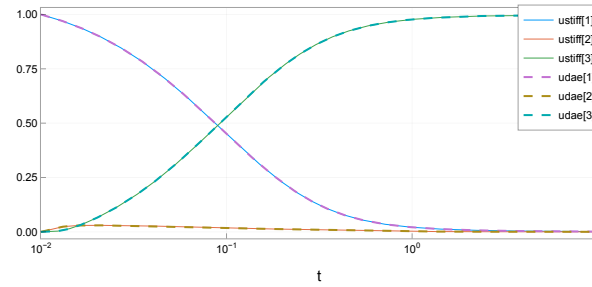
- udae_deq=let
-   fdae=ODEFunction(fdae!, mass_matrix=Diagonal([1, 1, 0]))
-   prob=ODEProblem(fdae, [1.0, 0, 0], (1.0e-2, 10), (k1=10, k2=10*logst, k3=100))
-   solve(prob, diffeqmethods[stmethod](); reltol=10*logstreltol);
- end;

```

(timesteps = 44, smallest = 1.0e-6, largest = 2.01611)

Method: Rosenbrock23 (Rosenbrock) ▾

Stiffness  $\log(d)$  =  4.0  $\log(\text{reltol})$ :  -2.0



## Conclusions

- There is a whole world of ODE and DAE solvers. DifferentialEquations.jl covers a significant part of this world
- ODE systems created from PDE discretizations are *large* and they are *stiff*, i.e. they contain strongly differing timescales
- Stability of explicit methods is limited by that of the fastest timescale, they are not feasible for stiff problems
- Explicit methods do not work for DAEs by construction
- A- and L- stable methods are required for stiff problems

The general advise when solving a particular problem is to use the capabilities of DifferentialEquations.jl and to find out which methods work best

### Helper cells

### Table of Contents

#### ODE and DAE Solvers

- Explicit Euler method
  - Test problem
- $\theta$ - and Runge-Kutta methods
  - $\theta$  method: implementation
  - Runge-Kutta methods
- Stability
  - A-Stability
  - L-Stability
- Approximation order
- An ODE system
- Stiff problems
- DAE problems
- Multistep methods
  - BDF Methods
  - Adams methods
- DifferentialEquations.jl
  - Lotka-Volterra revisited
  - Stiffness + DAE revisited
- Conclusions

