ODE and DAE Solvers

Advanced Topics from Scientific Computing
TU Berlin Winter 2024/25
Notebook 09
Jürgen Fuhrmann

```
GRBackend()
```

```
begin
    using LinearAlgebra: norm, I, Diagonal
    using ForwardDiff: ForwardDiff
    using DiffResults: DiffResults
    using NLsolve: nlsolve, converged
    using RecursiveArrayTools: DiffEqArray
    using Plots: Plots, plot, plot!, contourf
    using Plots: hline!, vline!, grid
    Plots.gr()
end
```

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 unpublisued sources used:

- TU Chemnitz lecture notes by A. Naumann
- 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 $u : [0,T] \to \mathbb{R}^m$ be a time dependent, differentiable function such that it fulfills the initial value problem for the ordinary differential equation (ODE) system:

$$\begin{cases} \dot{u} & = f(u, p, t) \\ u|_{t=0} & = u_0 \end{cases}$$

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

Here, p is a k-vector of parameters.

The system is called *autonomous* if 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 $u_n = u(t_n)$.

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.

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

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

```julia
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],
    ) 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 = told + dt
        push!(u, unew)
        push!(t, tnew)
    end
    return u, t
end;
```

Test problem

Let us first look at the equation

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

```
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 $t_{\frac{1}{2}}$

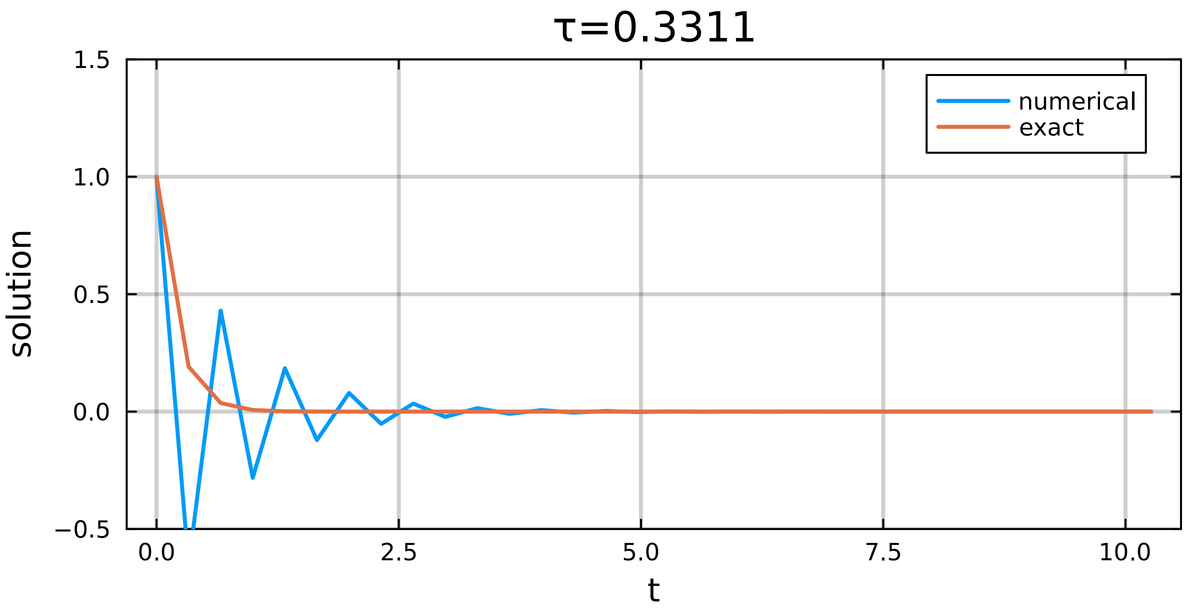
```
p_test = -5
```

```
p_test = -5
```

```
 (
     1:   [[1.0], [-0.655656], [0.429884], [-0.281856], [0.1848], [-0.121165], [0.079442
     2:   [0.0, 0.331131, 0.662262, 0.993393, 1.32452, 1.65566, 1.98679, 2.31792, 2.6490
)
```

```
u_test, t_test = explicit_euler(
    f_test!, p_test; u0 = [1.0], dt = τ1,
    tspan = (0, 10)
)
```

$\tau =$ 0.3311311214825911



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

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

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

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 implict Euler or linear implicit midpoint)
- it implements the solution of a slightly more general problem: for a matrix M (unit matrix by default), solve

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

```julia
function θmethod(
        f!::F, # right hand side function
        p; # Parameters
        u0 = [0.0], # Initial value
        dt = 0.1, # time step size
        tspan = (0, 1), # time intetval
        θ = 0, # θ parameter
        linear = false, # switch between linear method and newton solver
        M = I, # Mass matrix
        tol = 1.0e-10,
    ) where {F}
    # Start arrays for the result
    u = [u0]
    t = Float64[tspan[1]]
    uold = u[end]
    told = t[end]

    # Each timestep involves the solution of F(unew)=0
    # Here, unew is the  current iteration and F is the result
    function fstep!(F, unew)
        f!(F, θ * unew + (1 - θ) * uold, p, told + θ * dt)
        return F .= M * (unew .- 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)
            unew = uold -
                DiffResults.jacobian(diffresult) \
                DiffResults.value(diffresult)
        else
            # Solve time step problem and check for convergence
            result = nlsolve(fstep!, uold; autodiff = :forward, xtol = tol)
            !converged(result) && throw("convergence error")
            unew = result.zero
        end
        # Extend solution arrays
        push!(u, unew)
        push!(t, told + dt)
    end
    # Return the solution as DiffEqArray
    # from RecursiveArrayTools.jl (easy to plot)
    return DiffEqArray(u, t)
end;
```

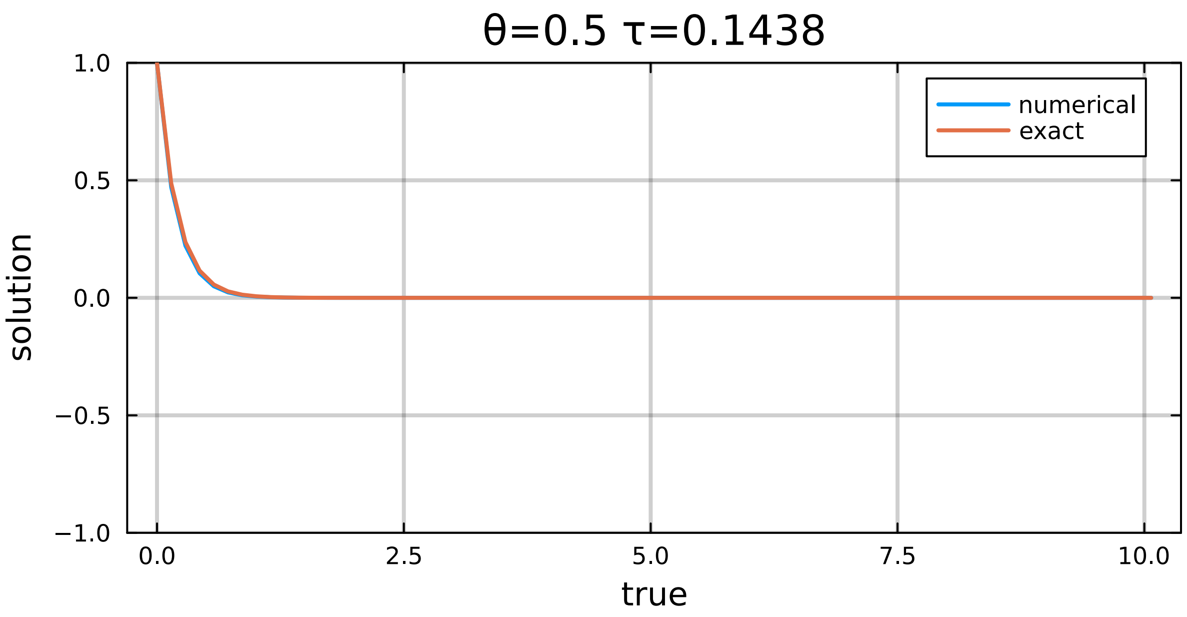
So let us use this for our test example:

u0_test =

	timestamp	value1
1	0.0	1.0
2	0.143845	0.471007
3	0.28769	0.221848
4	0.431535	0.104492
5	0.57538	0.0492165
6	0.719225	0.0231813
7	0.86307	0.0109186
8	1.00691	0.00514273
9	1.15076	0.00242227
10	1.2946	0.00114091
more		

```julia
u0_test = θmethod(f_test!, p_test; u0 = [1.0], dt = τ2, tspan = (0, 10), θ = θx)
```

θ= 0.5 τ= 0.14384498882876628



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 tableaus*:

c	A
	b

These tableaus can be conveniently manages using the `RungeKutta.jl` package.

```julia
begin
    using RungeKutta: RungeKutta, Tableau
    using RungeKutta: TableauExplicitEuler, TableauImplicitEuler
    using RungeKutta: TableauImplicitMidpoint, TableauRK4
    using RungeKutta: TableauRadauIIA
end
```

The θ method is a Runge-Kutta method:

```
Runge-Kutta Tableau ExplicitEuler with 1 stage and order 1:

 0//1 │ 0//1
──────┼──────
      │ 1//1
```

```julia
TableauExplicitEuler(Rational)
```

```
Runge-Kutta Tableau ImplicitEuler with 1 stage and order 1:

 1//1 │ 1//1
──────┼──────
      │ 1//1
```

```
TableauImplicitEuler(Rational)
```

```
Runge-Kutta Tableau ImplicitMidpoint with 1 stage and order 2:

 1//2 │ 1//2
──────┼──────
      │ 1//1
```

```
TableauImplicitMidpoint(Rational)
```

The classical Runge-Kutta method is "RK4":

```
Runge-Kutta Tableau RK416 with 4 stages and order 4:

 0//1 │ 0//1  0//1  0//1  0//1
 1//2 │ 1//2  0//1  0//1  0//1
 1//2 │ 0//1  1//2  0//1  0//1
 1//1 │ 0//1  0//1  1//1  0//1
──────┼────────────────────────
      │ 1//6  1//3  1//3  1//6
```

```
TableauRK4(Rational)
```

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

```
TableauDP (generic function with 4 methods)
```

```julia
begin
    function TableauDP(T = Float64, order = 4)
        A = T[
            0 0 0 0 0 0 0
            1 // 5 0 0 0 0 0 0
            3 // 40 9 // 40 0 0 0 0 0
            44 // 45 -56 // 15 32 // 9 0 0 0 0
            19372 // 6561 -25360 // 2187 64448 // 6561 -212 // 729 0 0 0
            9017 // 3168 -355 // 33 46732 // 5247 49 // 176 -5103 // 18656 0 0
            35 // 384 0 500 // 1113 125 // 192 -2187 // 6784 11 // 84 0
        ]
        c = T[0; 1 // 5; 3 // 10; 4 // 5; 8 // 9; 1; 1]
        b4 = T[
            5179 // 57600
            0
            7571 // 16695
            393 // 640
            -92097 // 339200
            187 // 2100
            1 // 40
        ]
        b5 = T[
            35 // 384; 0; 500 // 1113; 125 // 192; -2187 // 6784; 11 // 84;
            0
        ]

        return if order == 4
            RungeKutta.Tableau(:DoPri4, order, A, b4, c)
        elseif order == 5
            RungeKutta.Tableau(:DoPri5, order, A, b5, c)
        else
            throw(error("DormandPrince can have orders 4 or 5"))
        end
    end
    TableauDP(o::Number) = TableauDP(Float64, o)
end
```

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

    0//1 │        0//1          0//1          0//1         0//1          0//1
    1//5 │        1//5          0//1          0//1         0//1          0//1
   3//10 │        3//40         9//40         0//1         0//1          0//1
    4//5 │       44//45       -56//15        32//9         0//1          0//1
    8//9 │  19372//6561   -25360//2187   64448//6561    -212//729          0//1
    1//1 │   9017//3168     -355//33     46732//5247      49//176    -5103//18656
    1//1 │     35//384          0//1       500//1113     125//192    -2187//6784
─────────┼──────────────────────────────────────────────────────────────────────────
         │     35//384          0//1       500//1113     125//192    -2187//6784

                                                                  2 columns omitt
```

```julia
TableauDP(Rational, 5)
```

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

```
Runge-Kutta Tableau RadauIIA 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),
    ) 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 in 1:s
            du .= 0
            for k in 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 in 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
    return DiffEqArray(u, t)
end
```

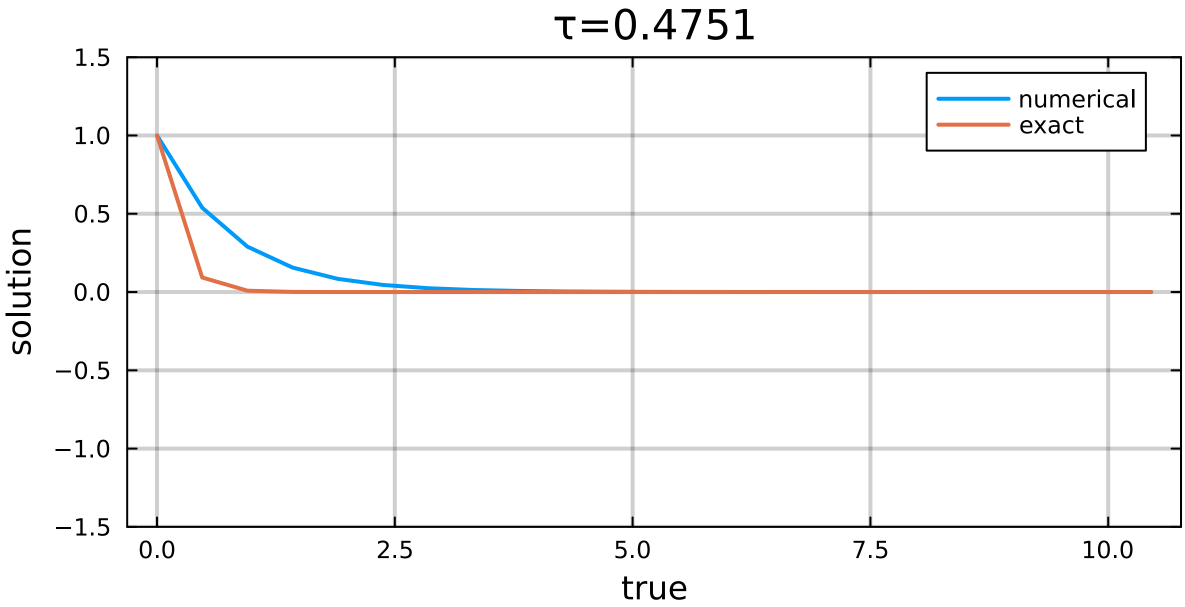
```
tableau = TableauRK4();
```

urk_test =

	timestamp	value1
1	0.0	1.0
2	0.475081	0.538578
3	0.950162	0.290066
4	1.42524	0.156223
5	1.90032	0.0841384
6	2.37541	0.0453151
7	2.85049	0.0244057
8	3.32557	0.0131444
9	3.80065	0.00707927
10	4.27573	0.00381274
	more	

```
urk_test = explicit_rk(
    f_test!, p_test; u0 = [1.0], tableau, dt = τrk,
    tspan = (0, 10)
)
```

τ= 0.47508101621027965



```
let
    p = plot(;
        framestyle = :box,
        gridlinewidth = 2,
        size = (600, 300),
        xlabel = "t",
        ylabel = "solution", ylimits = (-1.5, 1.5),
        title="τ=$(round(τrk, sigdigits=4))",
    )
    plot!(p, urk_test; linewidth = 2, label = "numerical")
    T = urk_test.t
    plot!(p, T, exact_test.(1.0, p_test, T); label = "exact", linewidth = 2)
end
```

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 ⇒ stability
- How good is the approximation ⇒ approximation order

Stability

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

Accidentally, the test problem $\dot{u} = \lambda u$ 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 $\lambda \in \mathbb{C}$ with $\mathrm{Re}\lambda < 0$, the approximate solution of the Dahlquist test equation fulfills

$|u_{n+1}| \leq |u_n|$.

The method is called conditionally A stable if $|u_{n+1}| \leq |u_n|$ is fulfilled for $h\lambda \in S \subset \mathbb{C}$, where S is called stability region.

In order to investigate A-Stability, it is useful to define the stability function R such that $u_{n+1} = R(\tau\lambda)(u_n)$

For the θ-methods this gives:

```julia
R_0(z) = 1 + z;
```

```julia
R_05(z) = (1 + z / 2) / (1 - z / 2);
```

```julia
R_1(z) = 1 / (1 - z);
```

For generic Runge-Kutta methods, we have:

```julia
function R_rk(z, tableau = TableauRK4())
    A = tableau.a
    b = tableau.b
    Ib = ones(length(b))
    return 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 $\mathrm{Re}\lambda < 0$, the solution of the Dahlquist test problem fulfils

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

independent of the stepsize τ.

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

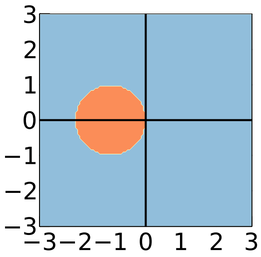
$$\lim_{\tau \to \infty} R(\tau\lambda) = 0$$

We can visualize the stability region by plotting the range where R<1 in red:

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

```
function stabregion(
        R;
        xrange = [-3, 3],
        yrange = [-3, 3],
        size = (150, 150),
        nx = 100,
        ny = 100,
        linewidth = 1,
    )
    X = range(xrange...; length = nx)
    Y = range(yrange...; length = ny)
    p = contourf(
        X,
        Y,
        (x, y) -> abs(R(x + y * im)) < 1 ? -1 : 3;
        levels = -1:1.0:2,
        colormap = :RdYlBu_3,
        legend = :none,
        xrange,
        yrange,
        linewidth = 0,
        size,
        gridalpha = 1,
        gridlinewidth = 1,
        gridstyle = :solid,
        aspect_ratio = 1.0,
    )
    hline!(p, [0]; color = :black)
    vline!(p, [0]; color = :black)
    p
end
```

Stability of the theta methods:

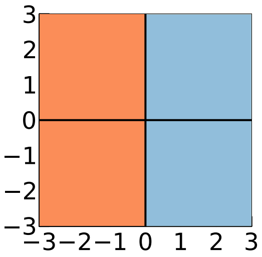


Explicit Euler

$R(z) = 1 + z \overset{z\to\infty}{\to} \infty$

Conditionally A-stable

not L-stable

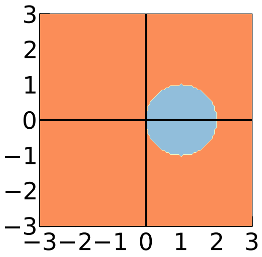


Implicit Midpoint

$R(z) = \frac{2+z}{2-z} \overset{z\to\infty}{\to} -1$

A-stable

not L-stable



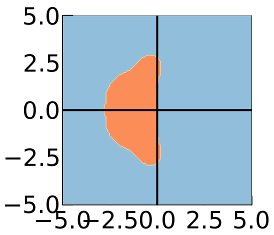
Implicit Euler

$R(z) = \frac{1}{1-z} \overset{z\to\infty}{\to} 0$

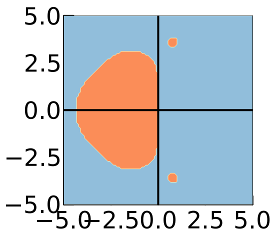
A-stable

L-stable

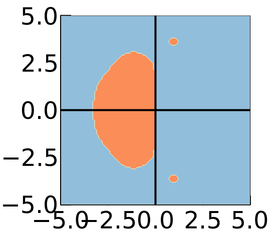
Explicit Runge-Kutta methods are conditionally A stable:



RK4

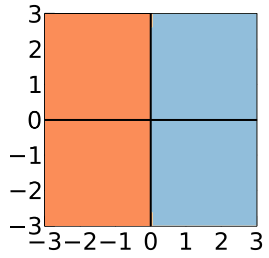


DP4

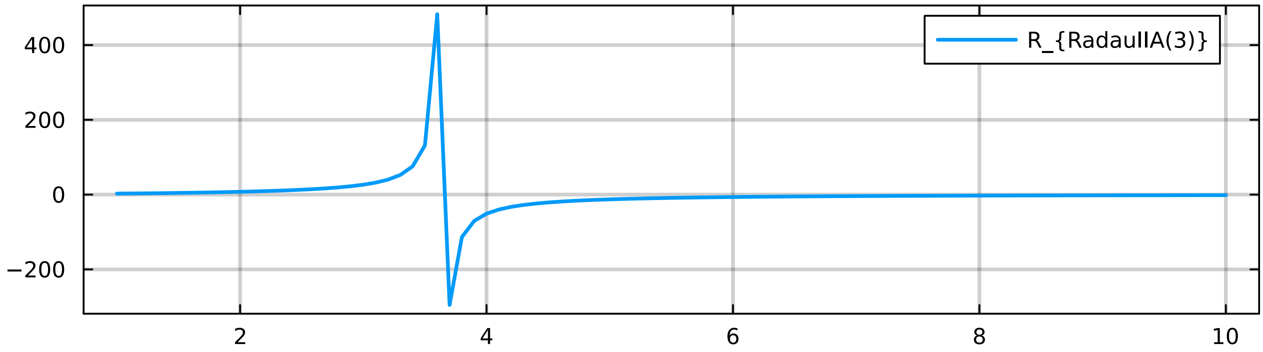


DP5

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



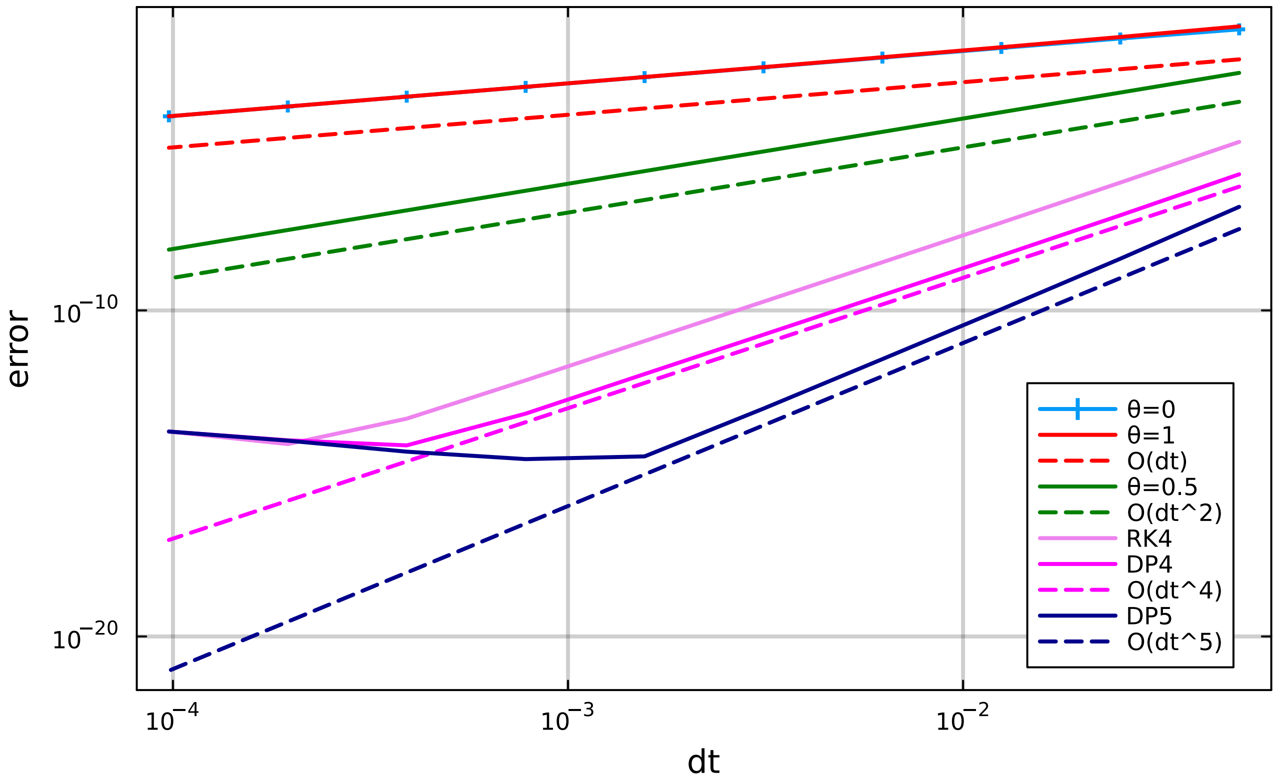
```
stabregion(z -> R_rk(z, TableauRadauIIA(3)))
```



Approximation order

Approximation order can generally can 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 demonstate the Lotka-Volterra system describing predator-prey dynamics:

- species x: prey whith growth rate a being eaten with rate by by the predators
- species y: predators growing with rate dx 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
    return du[2] = -c * y + d * x * y
end
```

The function V is constant along trajectories – it es 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)
    return d * x - c * log(x) + b * y - a * log(y)
end;
```

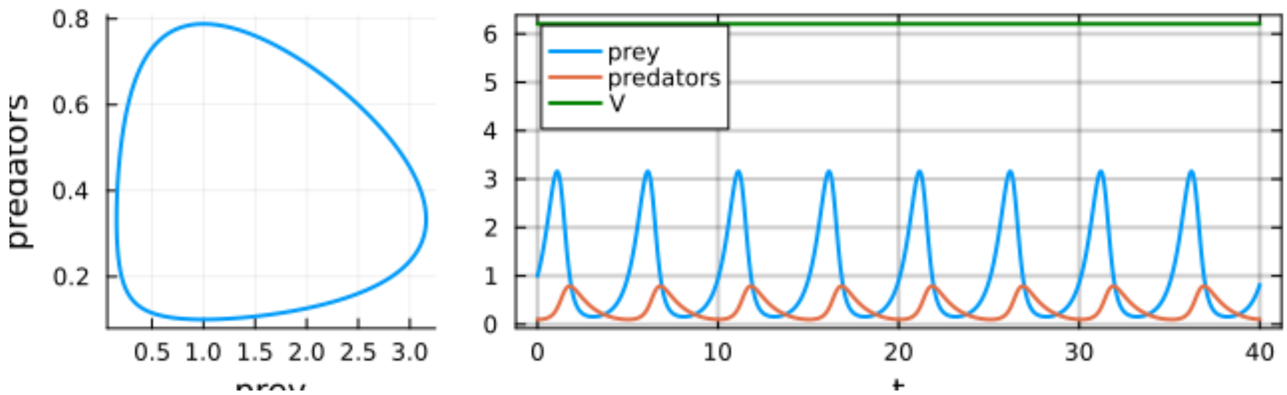
```
param =  (a = 2, b = 6, c = 1, d = 1)
```

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

```
ulv = if rklv
    explicit_rk(
        lotkavolterra!, param; u0 = [1.0, 0.1], dt = τlv,
        tspan = (0, 40)
    )
else
    θmethod(
        lotkavolterra!,
        param;
        u0 = [1.0, 0.1],
        dt = τlv,
        tspan = (0, 40),
        θ = θlv,
        linear = linlv,
    )
end;
```

linear= □ RK4= □

θ= 0.5 τ= 0.01



- 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 implict 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 κ 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)

$$
\begin{aligned}
u_1 &\xrightarrow{k_1} u_2\\
2u_2 &\xrightarrow{k_2} u_2 + u_3\\
u_2 + u_3 &\xrightarrow{k_3} u_1 + u_3
\end{aligned}
$$

```
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] - k₃ * u[2] * u[3] - k₂ * u[2]^2
    return 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.

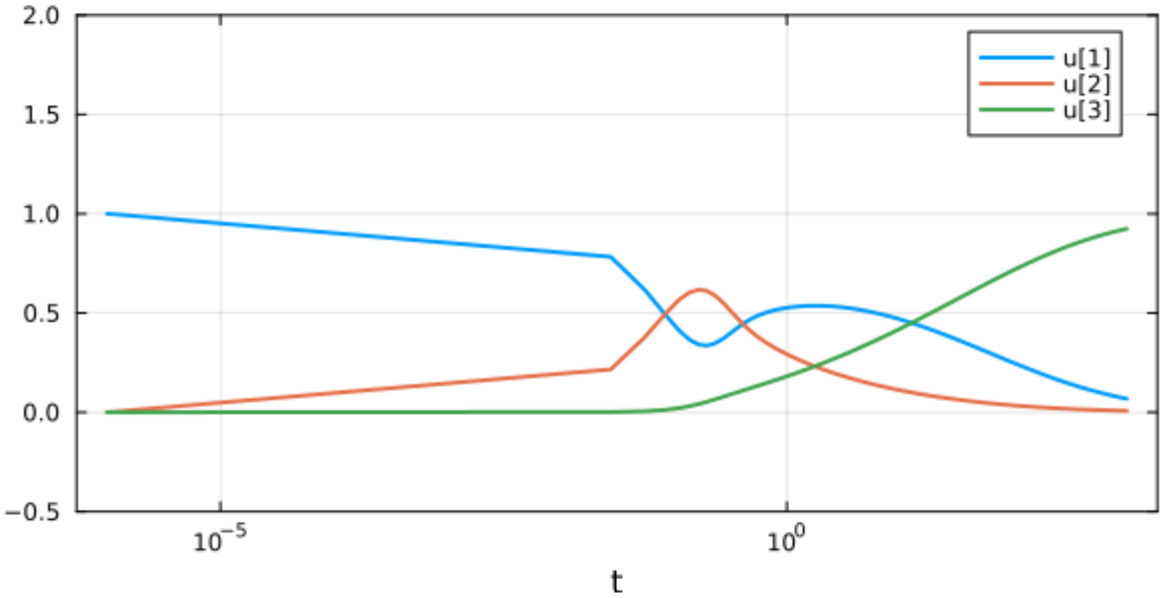
```
probertson =  (k₁ = 10, k₂ = 1.0, k₃ = 100)
```

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

```
ustiff = θmethod(
    fstiff!,
    probertson;
    u0 = [1.0, 0.0, 0.0],
    dt = τx2,
    θ = θx2,
    linear = lin2,
    tspan = (1.0e-6, 1.0e3)
);
```

θ= 1.0 τ= 0.027825594022071246 linear=
□

Stiffness k₂= 1.0



```
plot(
    ustiff;
    label = ["u[1]" "u[2]" "u[3]"],
    size = (600, 300),
    xlabel="t",
    linewidth = 2,
    framestyle = :box,
    xscale = :log10,
    ylimits=(-0.5,2)
)
```

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

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 correspondig 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 an L-stable methods.

Here, we re-define the Robertson example using the fact that the sum of all three reactants must be 1. This can be seen by adding up the three equations above resulting in `du[1]+du[2]+du[3]=0`. Thus `u[1]+u[2]+u[3]` must be constant and equal to the sum of the initial values.

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

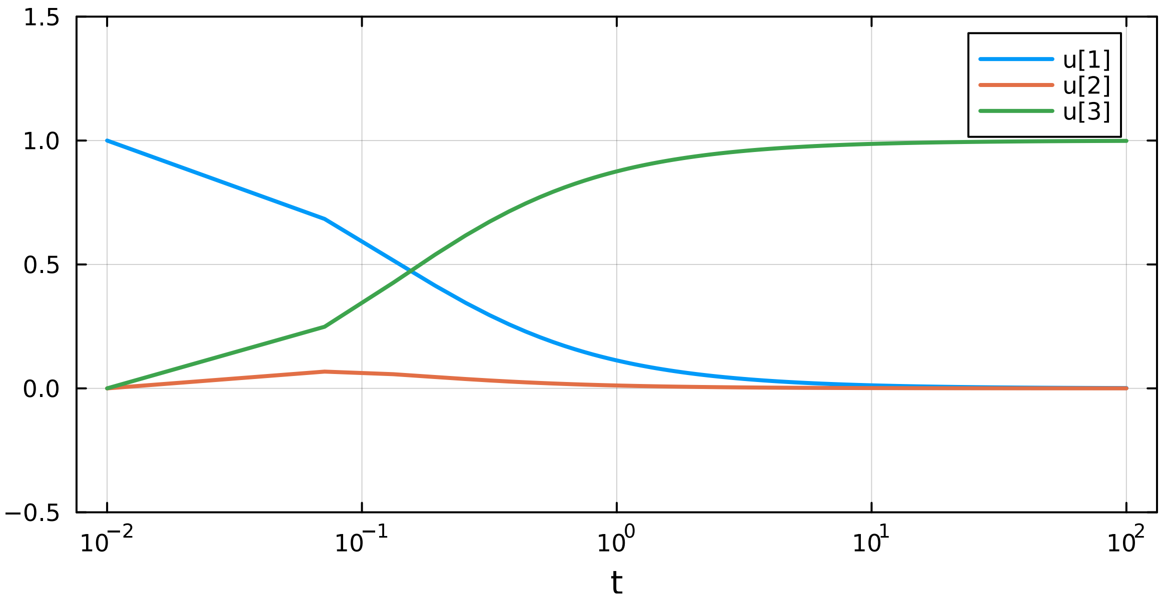
```
function fdae!(du, u, p, t)
    k₁, k₂, k₃ = p
    du[1] = -k₁ * u[1] + k₃ * u[2] * u[3]
    du[2] = k₁ * u[1] - k₃ * u[2] * u[3] - k₂ * u[2]^2
    du[3] = u[1] + u[2] + u[3] - 1
end
```

```
M_DAE = 3×3 Diagonal{Float64, Vector{Float64}}:
         1.0   ⋅    ⋅ 
          ⋅   1.0   ⋅ 
          ⋅    ⋅   0.0
```

```
M_DAE = Diagonal([1.0, 1, 0])
```

θ= 1.0 τ= 0.06135907273413173 linear= □

Stiffness k₂= 885.8667904100827



```
plot(
    udae;
    xlabel="t",
    label = ["u[1]" "u[2]" "u[3]"],
    size = (600, 300),
    linewidth = 2,
    framestyle = :box,
    xscale = :log10,
    ylimits=(-0.5,1.5)
)
```

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

Multistep methods

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

BDF Methods

For a given vector $X = x_1 \dots x_n$, and function values $Y = \{y_i = f(x_i)\}_{i=1...n}$ a Lagrange polynomial $L[X, Y]$ interpolates the function `f` in the interval $[x_1, x_n]$. It is given by

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

with

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

From the very definition we have $l_j[X](x_i) = \delta_{ij}$. The interesting point is that we can hope that $L'[X](x_n)$ is a good approximation of $f'(x_n)$. 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 $X = \{a, b, c\}$ we get

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

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

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

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

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.

Rosenbrock methods

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.

Differential equation solvers from sciml.ai

The SciML github org provides vast amount of differential equation solvers. These can be used by `using DifferentialEquations` providing all methods implemenented for all sorts of differential equations (ODE, stochastic DE, delay DE)

With `using OrdinaryDiffEq` one can obtain all ODE specific methods. For faster loading it is possible now to load more specific packages.

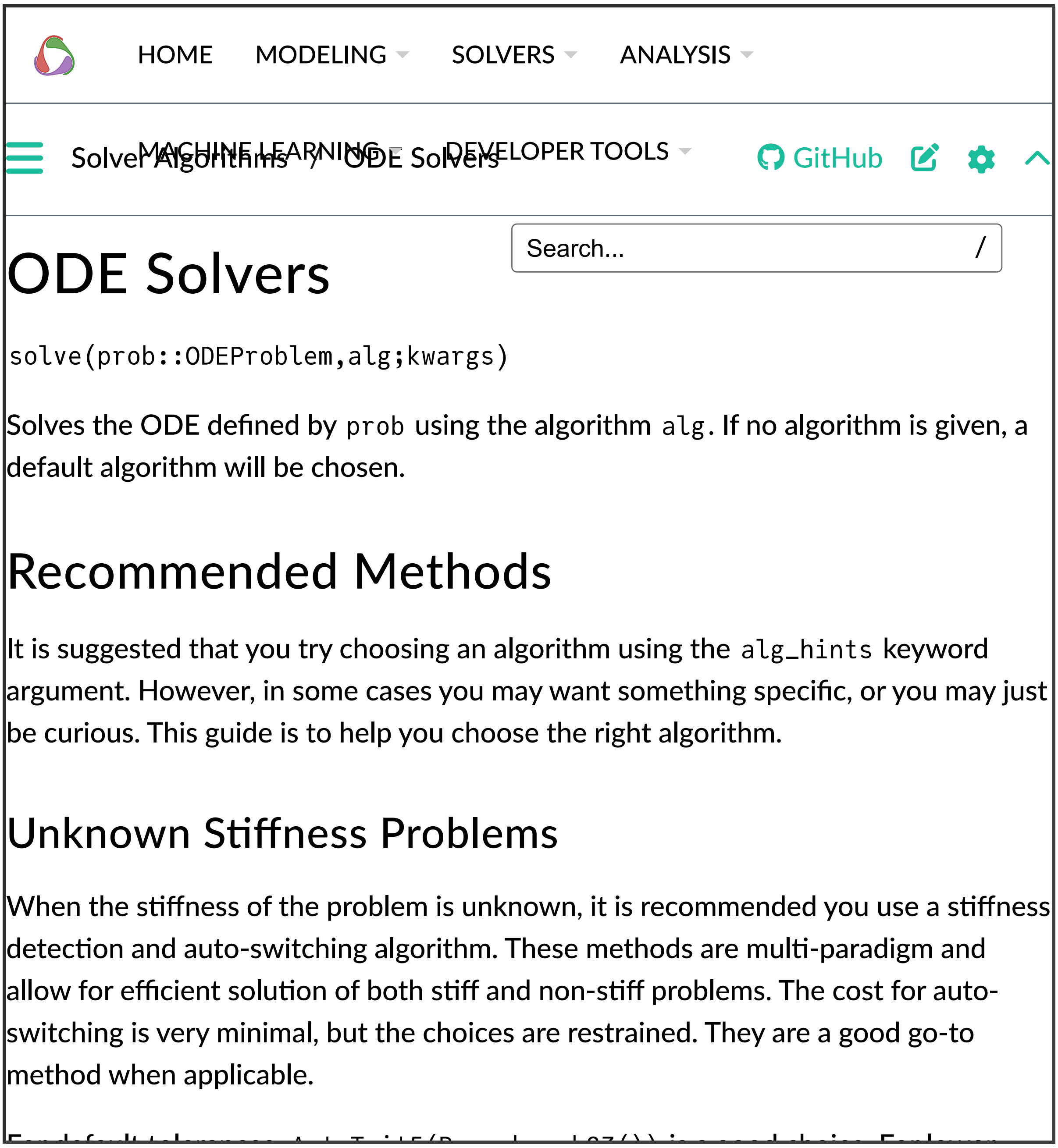
For an overview see the OrdinaryDiffEq documentation.

```julia
begin
    using SciMLBase: ODEProblem, ODEFunction, solve
    using OrdinaryDiffEqBDF: FBDF, QNDF2
    using OrdinaryDiffEqRosenbrock: Rosenbrock23
    using OrdinaryDiffEqLowOrderRK: RK4
    using OrdinaryDiffEqSDIRK: ImplicitEuler, ImplicitMidpoint
    using OrdinaryDiffEqFIRK: RadauIIA3
    using DataStructures: OrderedDict
end
```

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.

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 documention:



ODE Solvers

`solve(prob::ODEProblem,alg;kwargs)`

Solves the ODE defined by `prob` using the algorithm `alg`. If no algorithm is given, a default algorithm will be chosen.

Recommended Methods

It is suggested that you try choosing an algorithm using the `alg_hints` keyword argument. However, in some cases you may want something specific, or you may just be curious. This guide is to help you choose the right algorithm.

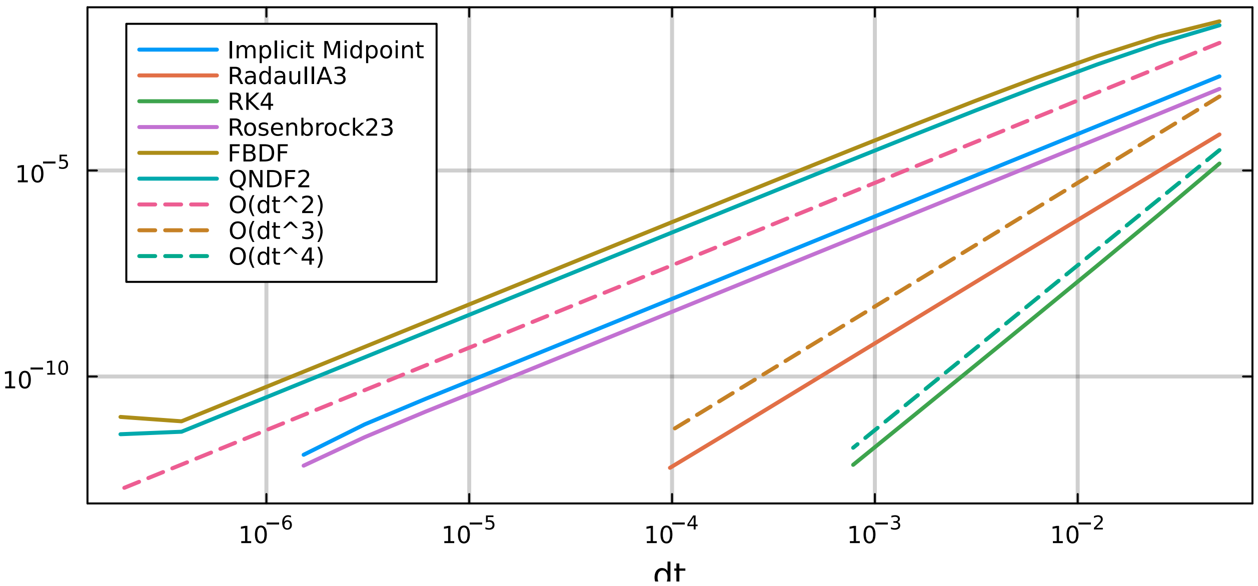
Unknown Stiffness Problems

When the stiffness of the problem is unknown, it is recommended you use a stiffness detection and auto-switching algorithm. These methods are multi-paradigm and allow for efficient solution of both stiff and non-stiff problems. The cost for auto-switching is very minimal, but the choices are restrained. They are a good go-to method when applicable.

For default tolerances, `AutoTsit5(Rosenbrock23())` is a good choice. For lower

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

```
diffeqmethods =   OrderedCollections.OrderedDict{String, UnionAll}(
                     "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)
```

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

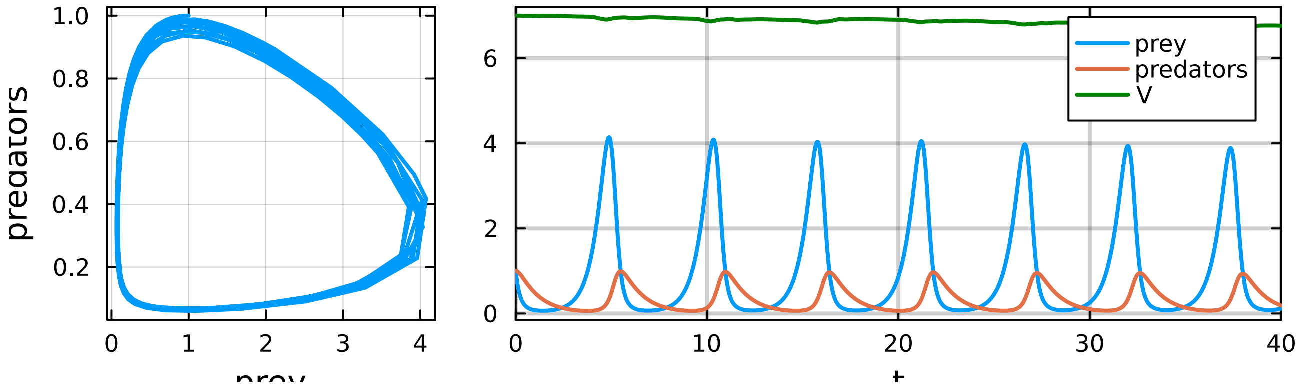
```julia
ulv_deq = let
    prob = ODEProblem(lotkavolterra!, [1.0, 1.0], (0, 40), param2)
    method = diffeqmethods[lvmethod]
    if method == ImplicitMidpoint
        dt = reltol
        @info "implicit midpoint: dt=$(dt)"
        solve(prob, method(); dt)
    else
        @info "reltol=$(reltol)"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

```
reltol=0.007054802310718643
```

method: FBDF ⌄ reltol|dt: 0.007054802310718643

```
(timesteps = 300, smallest = 6.0503e-6, largest = 0.268243)
```

```julia
analyze(ulv_deq)
```



```
let
    ulv = ulv_deq
    Vulv = DiffEqArray([V(ulv[:, i]..., param2) for i in 1:length(ulv)], ulv.t)
    puv = plot(
        ulv[1, :],
        ulv[2, :];
        linewidth = 2,
        framestyle = :box,
        size = (200, 200),
        label = nothing,
        xlabel = "prey",
        ylabel = "predators"
    )
    puvt = plot(
        ulv;
        linewidth = 2,
        framestyle = :box,
        gridlinewidth = 2,
        size = (400, 200),
        xlabel = "t",
        label = ["prey" "predators"]
    )
    plot!(puvt, Vulv; linewidth = 2, color = :green, label = "V")
    plot(puv, puvt; layout = grid(1, 2; widths = (0.3, 0.7)), size = (650, 200))
end
```

Stiffness + DAE revisited

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

```
(timesteps = 61, smallest = 1.3142e-5, largest = 1.78961)
```

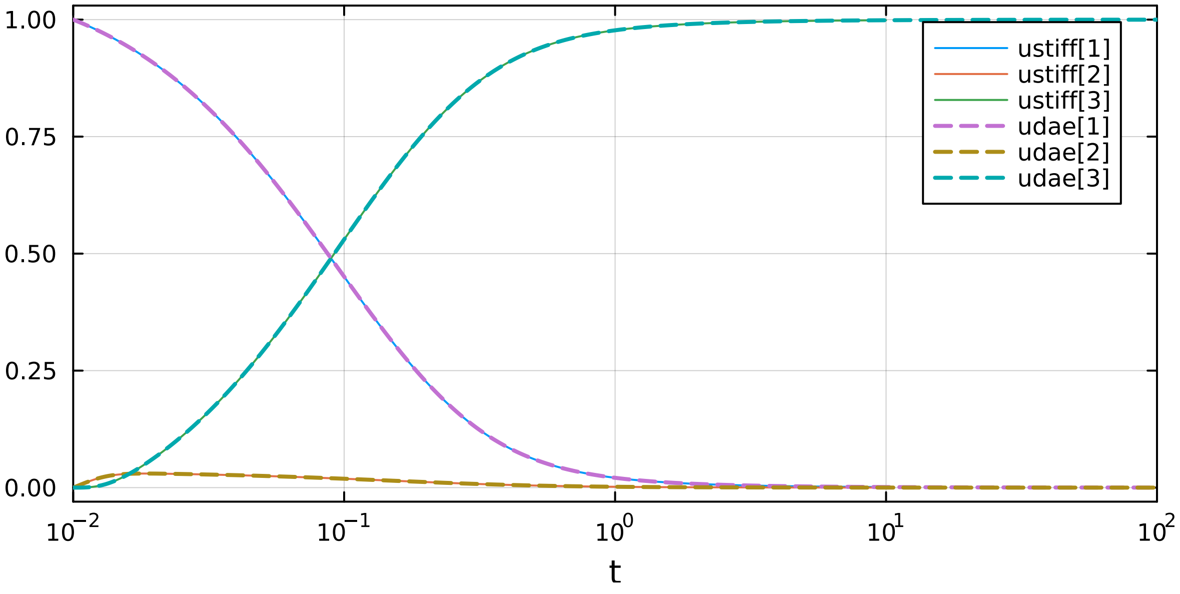
```
udae_deq = let
    fdae = ODEFunction(fdae!; mass_matrix = Diagonal([1, 1, 0]))
    prob = ODEProblem(
        fdae, [1.0, 0, 0], (1.0e-2, 100),
        (k₁ = 10, k₂ = st, k₃ = 100)
    )
    solve(prob, diffeqmethods[stmethod](); reltol = streltol)
end;
```

```
(timesteps = 66, smallest = 1.0e-6, largest = 22.0016)
```

Method: Rosenbrock23 (Rosenbrock)

Stiffness k₂= 10000.0

reltol: 0.0035111917342151313



Conclusions/outlook

- There is a whole world of ODE and DAE solvers. Julia's SciML solvers cover 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 Julia's SciML solvers and to find out which methods work best

---

Helper cells

Table of Contents

ODE and DAE Solvers

- Explicit Euler method
  - Test problem
- θ- and Runge-Kutta methods
  - θ 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
  - Rosenbrock methods
- Differential equation solvers from sciml.ai
  - Lotka-Volterra revisited
  - Stiffness + DAE revisited
- Conclusions/outlook

```
@statement_str (macro with 1 method)
```