Technical University of Denmark

02685 Scientific Computing for Differential Equations
2017

Assignment 1

Authors:
Miguel Suau de Castro (s161333)
Michal Baumgartner (s161636)

March 16, 2017

Contents

1 The Test Problem and DOPRI54 **2**
1.1 Explicit and Implicit Euler's method and Trapezoidal method . . . 2
1.2 Global and local errors 2
1.3 Error estimation 3

2 Design your own Explicit Runge-Kutta Method **4**
2.1 Order conditions, coefficients for the error estimator and the Butcher tableau 4
2.2 Testing on the test equation 5
2.3 Verifying the order 6
2.4 $\mathbf{R}(\lambda\mathbf{h})$ and stability plot 7
2.5 Testinging on the Van der Pol problem and comparison with `ode15s` 9

3 ESDIRK23 **9**
3.1 Implementation with fixed step size 9
3.2 Testing on Van der Pol problem and comparison with our designed ERK 10
3.3 Stability region, A and L-stability, practical implications 10
3.4 Implementation with variable step size and testing on Van der Pol problem 12

A ESDIRK23 fixed step size **13**

B ESDIRK23 variable step size **17**

1 The Test Problem and DOPRI54

In this first section we are going to implement a set of numerical methods for solving ordinary differential equations. Since the algorithms are only approximations to the real solution, we shall also test their accuracy and discuss their performance by comparing the results obtained when solving the two following initial value problems:

EQUATIONS

1.1 Explicit and Implicit Euler's method and Trapezoidal method

As a first approach, we are going to implement the Explicit Euler's method. The algorithm makes use of finite difference methods to replace the derivatives in the differential equation. The independent variable is discretized and the solution is computed based on coseqtive approximations to the real function values.

TALK ABOUT STEP LENGTH

EQUATION FORWARD EULER

Instead of using the previous iterate one could also look at future values to approximate a solution. This method is called backward or implicit Euler:

EQUATION BACKWARD EULER

However, for some problems the solution of the previous equation may require the use of numerical solvers, and thus the algorithm becomes computationally more demanding than the explicit Euler's method. We shall see in the next section the advantange of using this method.

Besides, the trapezodial method can be seen as a combination of both methods:

DESCRIBE TRAPEZOIDAL

Figure **??** shows the solution of the two initial value problems given by explicit, implicit Euler and trapezoidal, along with the true solution.

1.2 Global and local errors

It is easy to see in figure **??**, especially in the graph on the right, that, since we base the solution at one point on previous approximations, the further the points are from the initial value the more inaccurate they become and the greater the distance to the true solution is. This distance is called global error, whilst the error made in every iteration is known as local error. As we will discuss later the latter is commonly used to classify different methods depending in their accuracy.

One could then derive the analytical expression of the solution for both problems and compute te local and global errors.

ANALYTICAL SOLUTION TO THE PROBLEMS.

GLOBAL.

Figure **??** represents the global error at time $t = 10$ made by the three implemented solvers for different step sizes. As expected, the size the global error decreases when increasing the number of points used in the approximation. Besides, figure **??** shows the global error from $t = 0$ to $t = 10$.

On the other hand the local error at time $t = t_0 + h$, can be computed as:

EQUATION

Figure **??** shows the local error at the first iteration. Again we see how the error decreases with the step size. Moreover, as the plots use logarithmic scale and the curves are approximately straight lines, we can conclude that there is an exponential dependence between local error and step size or in big O notation: $O(h^{p+1})$. The constant p is used to characterize different methods, thus we say that a method is order 2 when the local error is proportional toh^3. The dashed lines in figure **??** can be used to determined the order of the three solvers. That is, order 1 for Explicit and Implicit Euler and order 2 for the Trapezoidal method.

1.3 Error estimation

Considering that the algorithms are used to solve differential equations that are hard to derive analyticaly, calculate the exact error is not always possible.

An easy way to estimate the local error is called step doubling. The solution is computed for ... (performance). It turns out that estimate is proportional to the exact error and we can then

More sophisticated algorithms use embbeded methods of lower order to estimate the error. This secondary method will be closely related to the main algorithm so that they can share computations and thus be very efficient.

The local error estimates are plotted along with the true errors in figure **??** for different step sizes. Even though, they do not match the exact values for some of the methods, the estimates lie always above the true errors which means that they can be used as an upper bound. Besides, as we know that the estimates are proportional to the exact local errors their slope can be used to verify the order of accuracy.

2 Design your own Explicit Runge-Kutta Method

2.1 Order conditions, coefficients for the error estimator and the Butcher tableau

Using the excerpt from the book provided in the lecture 10 folder we will write up the order conditions for an embedded Runge-Kutta method with 3 stages. The solution will have order 3 and the embedded method used for error estimation will have order 2.

Firstly the Butcher tableau for our ERK will have the following schema (henceforth the upper triangular shape where the a_{ij} coefficients are 0 and $c_1 = 0$):

0	0	0	0
c_2	a_{21}	0	0
c_3	a_{31}	a_{32}	0
x	b_1	b_2	b_3
$\widehat{x}$	$\widehat{b}_1$	$\widehat{b}_2$	$\widehat{b}_3$
e	d_1	d_2	d_3

Table 1: Butcher tableau for ERK with 3 stages and embedded method

Order conditions (one for first order, one for second order and two for third order) derived from our Butcher tableau:

$$\mathcal{O}(h^1): \quad b^T e = 1 \qquad b_1 + b_2 + b_3 = 1 \qquad \tau_1 \rightarrow \bullet \tag{1a}$$

$$\mathcal{O}(h^2): \quad b^T C e = \frac{1}{2} \qquad \underbrace{b_1 c_1}_{0} + b_2 c_2 + b_3 c_3 = \frac{1}{2} \qquad \tau_2 \rightarrow : \tag{1b}$$

$$\mathcal{O}(h^3): \quad b^T C^2 e = \frac{1}{3} \qquad \underbrace{b_1 c_1^2}_{0} + b_2 c_2^2 + b_3 c_3^2 = \frac{1}{3} \qquad \tau_3 \rightarrow \because \tag{1c}$$

$$b^T A C e = \frac{1}{6} \qquad \underbrace{b_2 a_{21} c_1}_{0} + \underbrace{b_3 a_{31} c_1}_{0} + b_3 a_{32} c_2 = \frac{1}{6} \qquad \tau_4 \rightarrow \vdots \tag{1d}$$

values of c_2 and c_3 will be set to $\frac{1}{4}$ and 1 respecively. This leaves us with 6 unknown variables (3 as and 3 bs) and only 4 equations so we will add the so called consistency conditions in order for the system to be solvable.

$$c_2 = a_{21} \tag{1e}$$

$$c_3 = a_{31} + a_{32} \tag{1f}$$

Using Matlab to solve the system we get the following results:

$$b_1 = -\tfrac{1}{6},\ b_2 = \tfrac{8}{9},\ b_3 = \tfrac{5}{18},\ a_{21} = \tfrac{1}{4},\ a_{31} = -\tfrac{7}{5},\ a_{32} = \tfrac{12}{5}.$$

Next we will solve the system defined for second order embedded method with one first order and one second order condition where c_2 and c_3 are known thus giving 2 equations with 3 unknowns. In order to find a solution, $\widehat{b}_2$ is set to be $\frac{1}{2}$[1].

$$\widehat{b}_1 + \widehat{b}_2 + \widehat{b}_3 = 1 \tag{2a}$$

$$\widehat{b}_2 c_2 + \widehat{b}_3 c_3 = \frac{1}{2} \tag{2b}$$

The above system yields $\widehat{b}_1 = \frac{1}{8}$ and $\widehat{b}_3 = \frac{3}{8}$. Going back to the Butcher tableau we know that last row $e = (d_1, d_2, d_3)$ is just the difference of the previous two rows by definition.

$c_1 = 0$	0	0	0
$c_2 = \frac{1}{4}$	$1/4$	0	0
$c_3 = 1$	$-7/5$	$12/5$	0
x	$-1/6$	$8/9$	$5/18$
$\widehat{x}$	$1/8$	$1/2$	$3/8$
e	$-7/24$	$7/18$	$-7/72$

Table 2: Butcher tableau with error estimators for our method

2.2 Testing on the test equation

Figure 1 depicts designed method and the analytical solution for various step sizes (0.1, 0.01, 0.001) in the rows along with absolute error (difference between true value of the test equation and the result of the designed method), the maximum error is shown in the plot's title as well as in a reference line. From the previously mentioned figure it is clear that the designed method works as expected.

[1] According to the book excerpt given in Lecture 10 folder. Otherwise any real value < 1 could have been selected.

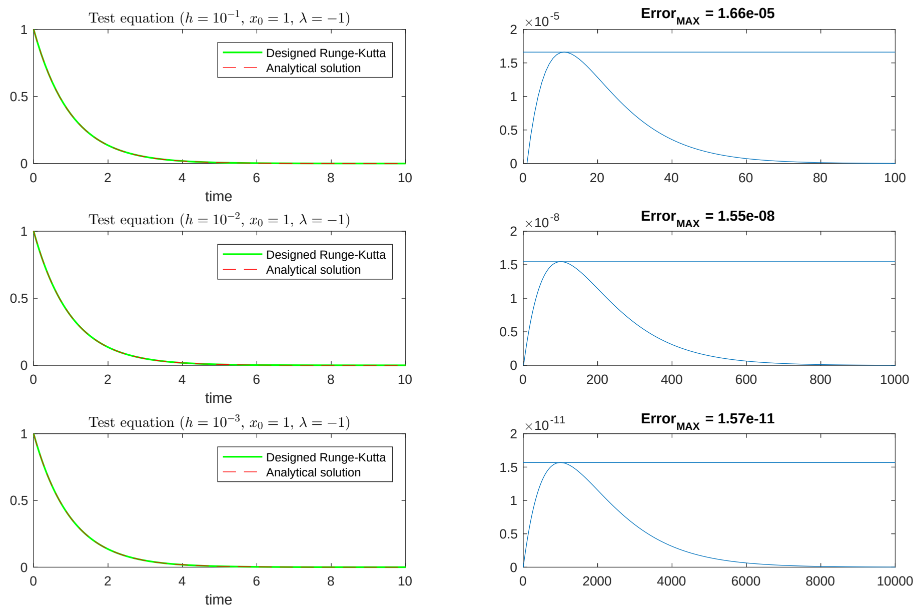

Figure 1: Comparison with the test equation for different step sizes

2.3 Verifying the order

Ten step sizes between 10^{-3} and 10^{-1} spaced logarithmicaly were chosen to plot the local error as a function of the step size. Loglog plot 2 along with dashed help lines is used in order to verify the order of the method designed. It can be seen that both entries are parallel with the help lines for $\mathcal{O}(h^3)$ and $\mathcal{O}(h^2)$ respectively, confirming that the method designed meets the order criteria specified in the beginning of this section.

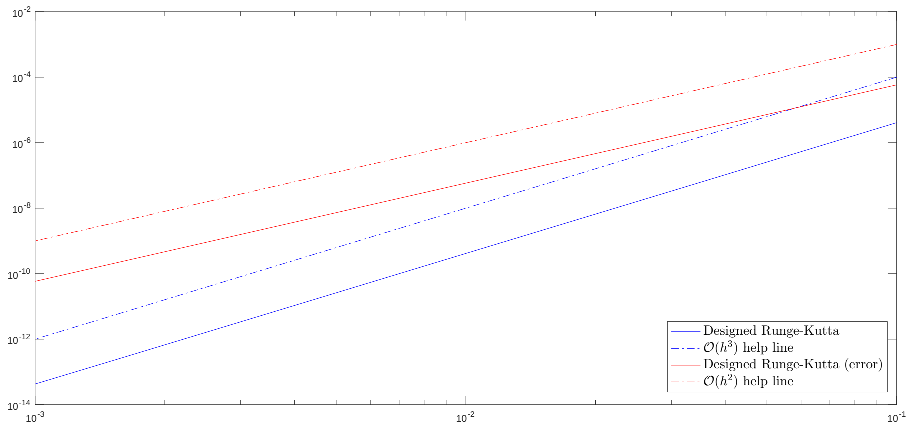

Figure 2: Loglog plot of the local error of designed ERK method (blue) and the returned error of the method (red, local error estimate) with help lines

2.4 R(λh) and stability plot

The solution to the test equation obtained by a Runge-Kutta method is defined as $x(t_n + h) = R(\lambda h)x(t_n)$ and $R(z) = 1 + zb^T(I - zA)^{-1}e$. From the Butcher tableau with error estimators for our method vector b and the A matrix are plugged in to $R(z)$ resulting in

$$R_m(z) = 1 + z + \frac{1}{2}z^2 + \frac{3}{18}z^3$$

where $z = \lambda h$ for the third order method. The second order embedded method yields

$$R_e(z) = z + \frac{1}{2}z^2 + \frac{9}{40}z^3$$

where $z = \lambda h$. Note that $R(z)$ can be calculated with Matlab's Symbolic Toolbox

```
syms z;
R = 1 + z*b'*inv(eye(length(b)) - z*A)*ones(length(b),1);
```

then `collect(R, z)` is used display powers of z and the respective coefficients.

The difference in stability of the designed and embedded method can be seen in figure 3. Although the plots look similar, the embedded method's stability region is slightly smaller, as well as all other metrics.

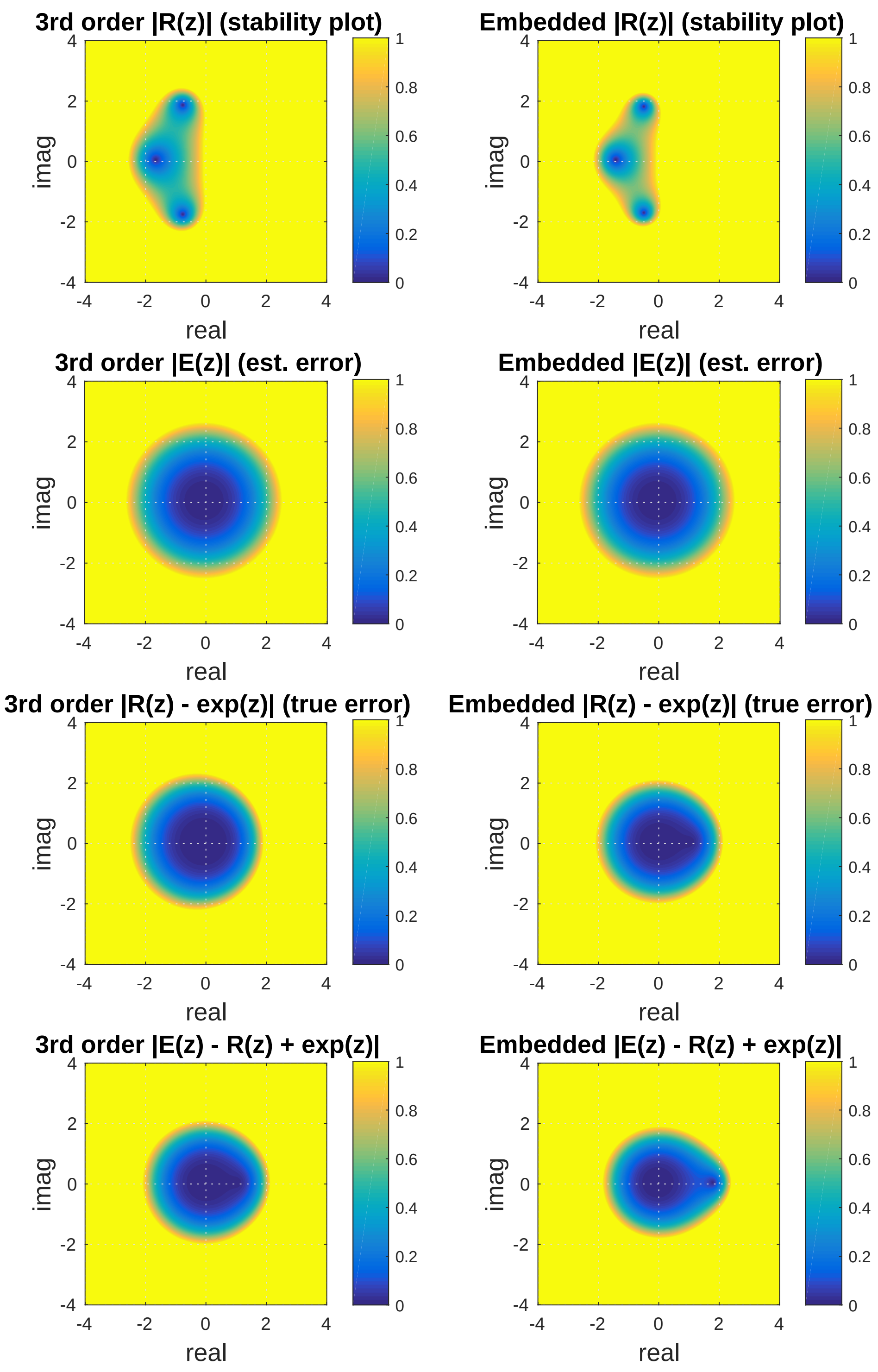

Figure 3: Stability plots of the third order ERK with second order embedded method. In order for the method to be A stable the whole left half plane has to be $|R(z)| < 1$ (i.e. in graphical representation shown not brightly yellow) which clearly it is not.

2.5 Testinging on the Van der Pol problem and comparison with `ode15s`

Matlab's `ode15s` was used with the default ODE-options and user defined Jacobian, the error for our method with with step size of 10^{-3} is roughly around 10^{-8} and for step size 10^{-2} it is around 10^{-5}. Even though our choice of $c_2 = 1/4$ might look strange, the method performs reasonably well.

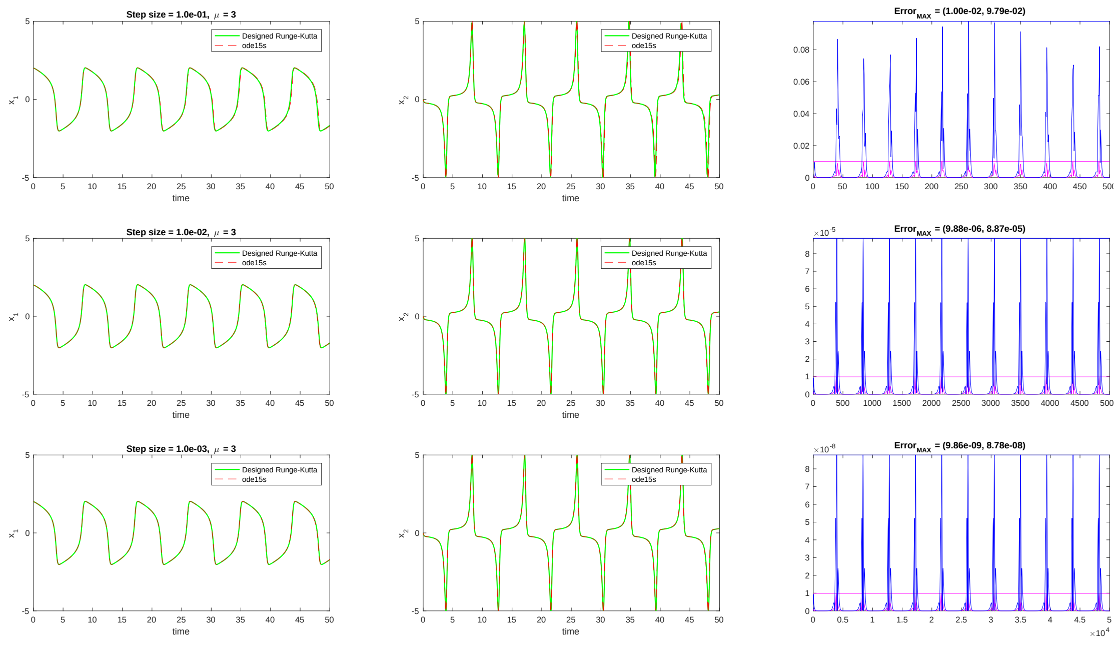

Figure 4: Comparison with `ode15s` on Van der Pol problem ($\mu = 3$). Each row depicts different step size (0.1, 0.01 and 0.001) and the maximal error from ERK is shown in the plot title as well as on a reference line (x_1 - magenta, x_2 - blue).

3 ESDIRK23

3.1 Implementation with fixed step size

Following the hint given at the lecture, the ESDIRK code from lecture files was used as a base to implement the fixed step size ESDIRK23. Given source is inspired by Tobias Ritschel's work on Numerical Methods For Solution of Differential Equations, however we aren't concerned about the modified version which uses function g as slightly modified initial value problem. Also fixed step size obviously doesn't require step size control so that part is removed.

Matlab code for this implementation is in the appendix.

3.2 Testing on Van der Pol problem and comparison with our designed ERK

The problem for the given μ of 100 is stiff as can be seen on figure 5 (second row) since the function values change a lot in very small timespan (mix of red and blue points). The step size was set to 0.001 and absolute and relative tolerances to 10^{-6}. Next we want to compare the previously designed ERK with ESDIRK23 in number of function evaluation in the method. For $\mu = 3$ the problem isn't stiff and ESDIRK23 has more function evaluations than ERK (60942 vs 60000), however when $\mu = 100$, ESDIRK23 computes 403695 evaluations versus 600000 of ERK resulting in almost 33% decrease and making ESDIRK23 better candidate for stiff problems.

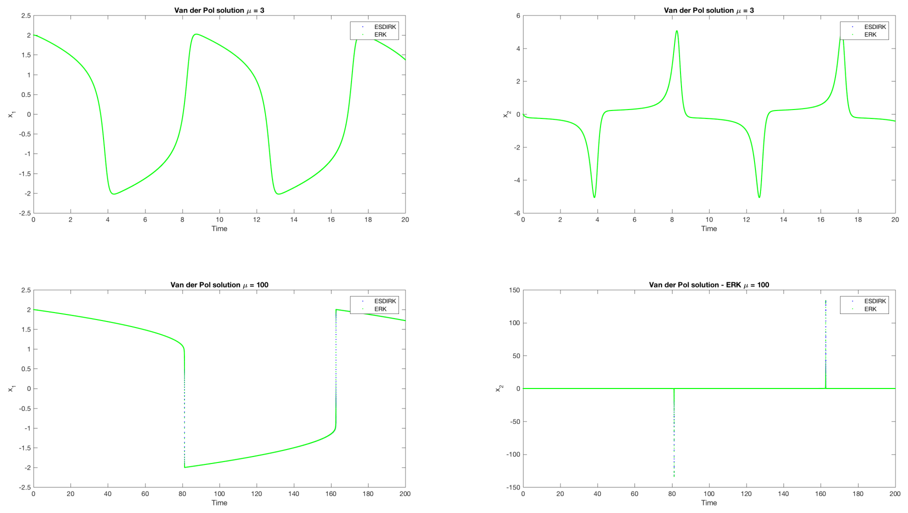

Figure 5: ESDIRK23 vs our ERK method on the Van der Pol problem with $\mu = 3$ and $\mu = 100$ (stiff)

3.3 Stability region, A and L-stability, practical implications

We can rewrite the stability function $R(z)$ as $R(z) = 1 + zb^T(I - zA)^{-1}e = \frac{\det(I-zA+zeb^T)}{\det(I-zA)}$ and using Mathematica or Matlab's Symbolic Toolbox explicitly calculate the numerator and denominator resulting in $R(z) = \frac{z-2z\gamma+1}{(\gamma z-1)^2} = \frac{1+z(1-2\gamma)}{(1-\gamma z)^2}$. L-stability is defined as $\lim_{z\to\infty} |R(z)| = 0$, again using Matlab we can verify that is holds or simply applying L'Hospital's rule once on the fraction yielding 0.

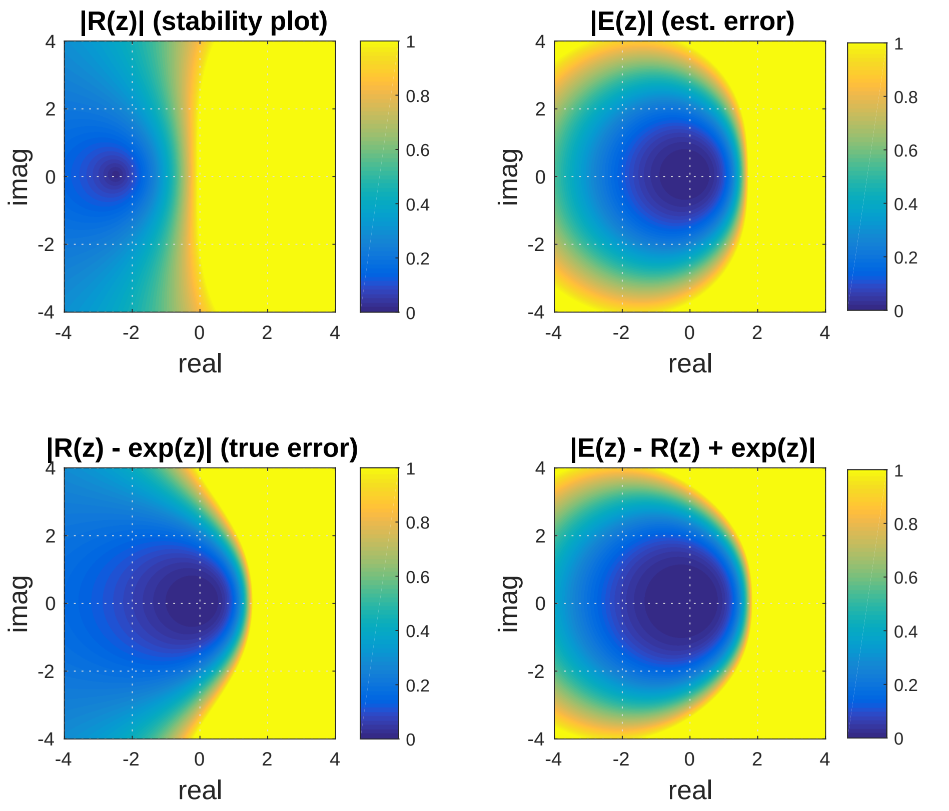

Figure 6: ESDIRK23 stability summary

Since the left half plane of $|R(z)| < 1$, we can conclude that the method is A-stable. From the Definition 8.2 in LeVeque we verified that this method is also L-stable (because it is A-stable and the previously mentioned limit is equal to 0). Even though the method is A and L-stable it's not suitable when $\lambda > 1$ because it is unstable as can be seen in the right halfplane of $|R(z)|$. Another thing worth noting is that the explicit Runge-Kutta methods (like question 3) have bounded stability region increasing with the order in contrast to implicit methods like ES-DIRK23 or the trapezoidal method.

3.4 Implementation with variable step size and testing on Van der Pol problem

As mentioned in the first part of this section the implementation is based on the given ESDIRK code in the lecture files folder. Similarly there is again no need for the g function as discussed before, but the step size control is kept and adapted to use the IVP in terms of $f(t, \mathbf{x})$ and $\mathbf{x}(t)$.

Firstly we step into the ”main” loop thar runs until we reach the final time and for all stages calculate the initial point (along with convergence check) and use Newton’s iterations (to obtain approximate X_i), which keep running only if we haven’t converged and we are not diverging as well as not converging too slowly, otherwise the iteration is halted. Then we decide based on convergence of the iterations whether to branch into error estimatation, step length controller (consequentialy accepting the step and using PI/Asymptotic controller for hr), perform convergence control for the step size and the Jacobian update strategy (evaluation and LU factorization) or just update the step size based on divergence or the freshness of Jacobian and calculate new LU factorization. In the end if the estimated error is small enough a step is taken.

Matlab code for this implementation is in the appendix.

A ESDIRK23 fixed step size

```matlab
function [Tout,Xout,info,stats] = ESDIRK23(fun,jac,t0,tf,x0,h0,absTol
    ,relTol,varargin)

% ESDIRK23
% Modified version for Ex5 according to the tips given in the lecture
    .
%
    ==========================================================================
% Runge-Kutta method parameters
gamma = 1-1/sqrt(2);
a31 = (1-gamma)/2;
AT = [0 gamma a31;0 gamma a31;0 0 gamma];
c  = [0; 2*gamma; 1];
b  = AT(:,3);
bhat = [    (6*gamma-1)/(12*gamma); ...
            1/(12*gamma*(1-2*gamma)); ...
            (1-3*gamma)/(3*(1-2*gamma))    ];
d  = b-bhat;
% p  = 2; % ex5
% phat = 3; % ex5
s = 3;


% error and convergence controller
epsilon = 0.8;
tau = 0.1*epsilon; %0.005*epsilon;
itermax = 20;
% ke0 = 1.0/phat;
% ke1 = 1.0/phat;
% ke2 = 1.0/phat;
% alpharef = 0.3;
% alphaJac = -0.2;
% alphaLU  = -0.2;
% hrmin = 0.01;
% hrmax = 10;
%
    =========================================================================
tspan = [t0 tf]; % carsten
info = struct(...
            'nStage',    s,        ... % carsten
            'absTol',    absTol,   ... % carsten % ex5
            'relTol',    relTol,   ... % carsten % ex5
            'iterMax',   itermax,  ... % carsten
            'tspan',     tspan,    ... % carsten
```

```
               'nFun',       0, ...
               'nJac',       0, ...
               'nLU',        0, ...
               'nBack',      0, ...
               'nStep',      0, ...
               'nAccept',    0, ...
               'nFail',      0, ...
               'nDiverge',   0, ...
               'nSlowConv', 0);



% Main ESDIRK Integrator
%
    ========================================================================

nx = size(x0,1);
F = zeros(nx,s);
t = t0;
x = x0;
h = h0;
IG = eye(length(x0)); % ex5 replacement for g

[F(:,1),~]  = feval(fun,t,x,varargin{:}); % ex5 no need for g
info.nFun = info.nFun+1;
[dfdx,~] = feval(jac,t,x,varargin{:}); % ex5 no need for g
info.nJac = info.nJac+1;
%FreshJacobian = true; % ex5
if (t+h)>tf
    h = tf-t;
end
hgamma = h*gamma;
dRdx = IG - hgamma*dfdx;
[L,U,pivot] = lu(dRdx,'vector');
info.nLU = info.nLU+1;
%hLU = h; % ex5

%FirstStep = true; % ex5
%ConvergenceRestriction = false; % ex5
%PreviousReject = false; % ex5
iter = zeros(1,s);

% Output
chunk = 100;
Tout = zeros(chunk,1);
Xout = zeros(chunk,nx);
%Gout = zeros(chunk,nx); % ex5

Tout(1,1) = t;
```

```
Xout(1,:) = x.';
%Gout(1,:) = g.'; % ex5

while t<tf
    info.nStep = info.nStep+1;
    %
        =====================================================================

    % A step in the ESDIRK method
    i=1;
    diverging = false;
    SlowConvergence = false; % carsten
    alpha = 0.0;
    Converged = true;
    while (i<s) && Converged
        % Stage i=2,...,s of the ESDIRK Method
        i=i+1;
        phi = x + F(:,1:i-1)*(h*AT(1:i-1,i)); % ex5

        % Initial guess for the state % ex5 removed duplicate code
        dt = c(i)*h;
        %G = g + dt*F(:,1);
        X = x + dt*F(:,1); % ex5
        T = t+dt;

        [F(:,i),~] = feval(fun,T,X,varargin{:}); % ex5
        info.nFun = info.nFun+1;
        R = X - hgamma*F(:,i) - phi; % ex5
        rNewton = norm(R./(absTol + abs(X).*relTol), inf); % ex5
        Converged = (rNewton < tau);

        % Newton Iterations
        while ~Converged && ~diverging && ~SlowConvergence
            iter(i) = iter(i)+1;
            dX = U\(L\(R(pivot,1)));
            info.nBack = info.nBack+1;
            X = X - dX;
            rNewtonOld = rNewton;
            [F(:,i),~] = feval(fun,T,X,varargin{:}); % ex5
            info.nFun = info.nFun+1;
            R = X - hgamma*F(:,i) - phi; % ex5
            rNewton = norm(R./(absTol + abs(X).*relTol), inf); % ex5
            alpha = max(alpha,rNewton/rNewtonOld);
            Converged = (rNewton < tau);
            diverging = (alpha >= 1);
            SlowConvergence = (iter(i) >= itermax);
        end
        diverging = (alpha >= 1)*i; % carsten, recording which stage
            is diverging
    end
    %if diverging, i, iter, pause, end
    nstep = info.nStep;
    stats.t(nstep) = t;
    stats.h(nstep) = h;
    stats.r(nstep) = NaN;
    stats.iter(nstep,:) = iter;
    stats.Converged(nstep) = Converged;
    stats.Diverged(nstep)  = diverging;
    stats.AcceptStep(nstep) = false;
    stats.SlowConv(nstep)  = SlowConvergence*i; % carsten, recording
       which stage is converging to slow (reaching maximum no. of
       iterations)
    iter(:) = 0; % carsten
    %
       ======================================================================

    % Error estimation
    e = F*(h*d);
    r = norm(e./(absTol + abs(X).*relTol), inf); % ex5
    r = max(r,eps);
    stats.r(nstep) = r;
    t = T;
    x = X;
    F(:,1) = F(:,s);

    % Jacobian Update Strategy
    [dfdx,~] = feval(jac,t,x,varargin{:}); % ex5
    info.nJac = info.nJac+1;
    hgamma = h*gamma;
    dRdx = IG - hgamma*dfdx; % ex5
    [L,U,pivot] = lu(dRdx,'vector');
    info.nLU = info.nLU+1;
    info.nFail = info.nFail + ~Converged; % ex5
    info.nDiverge = info.nDiverge + (~Converged && diverging); % ex5

    %
       ======================================================================

    % Storage of variables for output % ex5
    info.nAccept = info.nAccept + 1;
    nAccept = info.nAccept;
    if nAccept > length(Tout);
       Tout = [Tout; zeros(chunk,1)];
       Xout = [Xout; zeros(chunk,nx)];
    end
    Tout(nAccept,1) = t;
    Xout(nAccept,:) = x.';
```

```
end
info.nSlowConv = length(find(stats.SlowConv)); % carsten
Tout = Tout(1:nAccept,1);
Xout = Xout(1:nAccept,:);
```

B ESDIRK23 variable step size

```
function [Tout,Xout,info,stats] = ESDIRK23_Adaptive(fun,jac,t0,tf,x0,
    h0,absTol,relTol,varargin)

% ESDIRK23 Adaptive
% Modified for Ex5
%
    ==========================================================================
% Runge-Kutta method parameters
gamma = 1-1/sqrt(2);
a31 = (1-gamma)/2;
AT = [0 gamma a31;0 gamma a31;0 0 gamma];
c  = [0; 2*gamma; 1];
b  = AT(:,3);
bhat = [    (6*gamma-1)/(12*gamma); ...
            1/(12*gamma*(1-2*gamma)); ...
            (1-3*gamma)/(3*(1-2*gamma))    ];
d  = b-bhat;
p  = 2;
phat = 3;
s = 3;


% error and convergence controller
epsilon = 0.8;
tau = 0.1*epsilon; %0.005*epsilon;
itermax = 20;
ke0 = 1.0/phat;
ke1 = 1.0/phat;
ke2 = 1.0/phat;
alpharef = 0.3;
alphaJac = -0.2;
alphaLU  = -0.2;
hrmin = 0.01;
hrmax = 10;
%
    =========================================================================
tspan = [t0 tf]; % carsten
info = struct(...
            'nStage',    s,        ... % carsten
            'absTol',    absTol,   ... % carsten
            'relTol',    relTol,   ... % carsten
            'iterMax',   itermax,  ... % carsten
            'tspan',     tspan,    ... % carsten
            'nFun',      0, ...
            'nJac',      0, ...
            'nLU',       0, ...
            'nBack',     0, ...
            'nStep',     0, ...
            'nAccept',   0, ...
            'nFail',     0, ...
            'nDiverge',  0, ...
            'nSlowConv', 0);



% Main ESDIRK Integrator
%
    ==========================================================================

nx = size(x0,1);
F = zeros(nx,s);
t = t0;
x = x0;
h = h0;
IG = eye(length(x0)); % replaces g

[F(:,1),~]  = feval(fun,t,x,varargin{:}); % ex5
info.nFun = info.nFun+1;
[dfdx,~] = feval(jac,t,x,varargin{:}); % ex5
info.nJac = info.nJac+1;
FreshJacobian = true;
if (t+h)>tf
    h = tf-t;
end
hgamma = h*gamma;
dRdx = IG - hgamma*dfdx; % ex5
[L,U,pivot] = lu(dRdx,'vector');
info.nLU = info.nLU+1;
hLU = h;

FirstStep = true;
ConvergenceRestriction = false;
PreviousReject = false;
iter = zeros(1,s);

% Output
chunk = 100;
Tout = zeros(chunk,1);
```

```matlab
Xout = zeros(chunk,nx);

Tout(1,1) = t;
Xout(1,:) = x.';

while t<tf
    info.nStep = info.nStep+1;
    % =====================================================================
    % A step in the ESDIRK method
    i=1;
    diverging = false;
    SlowConvergence = false; % carsten
    alpha = 0.0;
    Converged = true;
    while (i<s) && Converged
        % Stage i=2,...,s of the ESDIRK Method
        i=i+1;
        phi = x + F(:,1:i-1)*(h*AT(1:i-1,i)); % ex5

        % Initial guess for the state
        dt = c(i)*h;
        X  = x + dt*F(:,1); % ex5
        T = t+dt;

        [F(:,i),~] = feval(fun,T,X,varargin{:}); % ex5
        info.nFun = info.nFun+1;
        R = X - hgamma*F(:,i) - phi; % ex5
        rNewton = norm(R./(absTol + abs(X).*relTol), inf);
        Converged = (rNewton < tau);

        % Newton Iterations
        while ~Converged && ~diverging && ~SlowConvergence
            iter(i) = iter(i)+1;
            dX = U\(L\(R(pivot,1)));
            info.nBack = info.nBack+1;
            X = X - dX;
            rNewtonOld = rNewton;
            [F(:,i),~] = feval(fun,T,X,varargin{:}); % ex5
            info.nFun = info.nFun+1;
            R = X - hgamma*F(:,i) - phi; % ex5
            rNewton = norm(R./(absTol + abs(X).*relTol), inf);
            alpha = max(alpha,rNewton/rNewtonOld);
            Converged = (rNewton < tau);
            diverging = (alpha >= 1);
            SlowConvergence = (iter(i) >= itermax); % carsten
        end
        diverging = (alpha >= 1)*i; % carsten, recording which stage
```

```
                is diverging
    end

    nstep = info.nStep;
    stats.t(nstep) = t;
    stats.h(nstep) = h;
    stats.r(nstep) = NaN;
    stats.iter(nstep,:) = iter;
    stats.Converged(nstep) = Converged;
    stats.Diverged(nstep)  = diverging;
    stats.AcceptStep(nstep) = false;
    stats.SlowConv(nstep)  = SlowConvergence*i; % carsten, recording
        which stage is converging to slow (reaching maximum no. of
        iterations)
    iter(:) = 0; % carsten
    %
        ======================================================================

    % Error and Convergence Controller
    if Converged
        % Error estimation
        e = F*(h*d);
        r = norm(e./(absTol + abs(X).*relTol), inf);
        CurrentStepAccept = (r<=1.0);
        r = max(r,eps);
        stats.r(nstep) = r;
        % Step Length Controller
        if CurrentStepAccept
            stats.AcceptStep(nstep) = true;
            info.nAccept = info.nAccept+1;
            if FirstStep || PreviousReject || ConvergenceRestriction
                % Aymptotic step length controller
                hr = 0.75*(epsilon/r)^ke0;
            else
                % Predictive controller
                s0 = (h/hacc);
                s1 = max(hrmin,min(hrmax,(racc/r)^ke1));
                s2 = max(hrmin,min(hrmax,(epsilon/r)^ke2));
                hr = 0.95*s0*s1*s2;
            end
            racc = r;
            hacc = h;
            FirstStep = false;
            PreviousReject = false;
            ConvergenceRestriction = false;

            % Next Step
            t = T;
            x = X;
```

```
                F(:,1) = F(:,s);

        else % Reject current step
            info.nFail = info.nFail+1;
            if PreviousReject
                kest = log(r/rrej)/(log(h/hrej));
                kest = min(max(0.1,kest),phat);
                hr   = max(hrmin,min(hrmax,((epsilon/r)^(1/kest))));
            else
                hr = max(hrmin,min(hrmax,((epsilon/r)^ke0)));
            end
            rrej = r;
            hrej = h;
            PreviousReject = true;
        end

        % Convergence control
        halpha = (alpharef/alpha);
        if (alpha > alpharef)
            ConvergenceRestriction = true;
            if hr < halpha
                h = max(hrmin,min(hrmax,hr))*h;
            else
                h = max(hrmin,min(hrmax,halpha))*h;
            end
        else
            h = max(hrmin,min(hrmax,hr))*h;
        end
        h = max(1e-8,h);
        if (t+h) > tf
            h = tf-t;
        end

        % Jacobian Update Strategy
        FreshJacobian = false;
        if alpha > alphaJac
            [dfdx,~] = feval(jac,t,x,varargin{:}); % ex5
            info.nJac = info.nJac+1;
            FreshJacobian = true;
            hgamma = h*gamma;
            dRdx = IG - hgamma*dfdx;  % ex5
            [L,U,pivot] = lu(dRdx,'vector');
            info.nLU = info.nLU+1;
            hLU = h;
        elseif (abs(h-hLU)/hLU) > alphaLU
            hgamma = h*gamma;
            dRdx = IG - hgamma*dfdx; % ex5
            [L,U,pivot] = lu(dRdx,'vector');
            info.nLU = info.nLU+1;
            hLU = h;
        end
    else % not converged
        info.nFail=info.nFail+1;
        CurrentStepAccept = false;
        ConvergenceRestriction = true;
        if FreshJacobian && diverging
            h = max(0.5*hrmin,alpharef/alpha)*h;
            info.nDiverge = info.nDiverge+1;
        elseif FreshJacobian
            if alpha > alpharef
                h = max(0.5*hrmin,alpharef/alpha)*h;
            else
                h = 0.5*h;
            end
        end
        if ~FreshJacobian
            [dfdx,~] = feval(jac,t,x,varargin{:}); % ex5
            info.nJac = info.nJac+1;
            FreshJacobian = true;
        end
        hgamma = h*gamma;
        dRdx = IG - hgamma*dfdx; % ex5
        [L,U,pivot] = lu(dRdx,'vector');
        info.nLU = info.nLU+1;
        hLU = h;
    end

    % ======================================================================

    % Storage of variables for output

    if CurrentStepAccept
       nAccept = info.nAccept;
       if nAccept > length(Tout);
           Tout = [Tout; zeros(chunk,1)];
           Xout = [Xout; zeros(chunk,nx)];
       end
       Tout(nAccept,1) = t;
       Xout(nAccept,:) = x.';
    end
end
info.nSlowConv = length(find(stats.SlowConv)); % carsten
nAccept = info.nAccept;
Tout = Tout(1:nAccept,1);
Xout = Xout(1:nAccept,:);
```