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

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:

$$\dot{x} = \lambda x(t) \qquad x(0) = 1 \qquad \lambda = -1 \tag{1}$$

$$\ddot{x} = -x(t) \qquad x(0) = 1 \qquad \dot{x(0)} = 0 \tag{2}$$

1.1 One step methods

As a first approach, we will apply Explicit Euler's method. This basic algorithm makes use of finite difference methods to replace the derivatives in the differential equation.

Considering a differential equation of the form:

$$\frac{dy}{dt} = f(t_n, y_n) \qquad y(t_0) = y_0 \tag{3}$$

If the independent variable is discretized, the solution can be obtained in n steps, by performing cosequtive approximations to the real function values.

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{4}$$

Where h is then the size of the step. It is easy to show that the smaller the step size is the more accurate our solution will be.

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:

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}) \tag{5}$$

However, for nonlinear problems the solution of the previous equation may require the use of numerical solvers, such as Newton's method, 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.

Finally, the trapezodial method can be seen as a combination of both methods, where the solution is computed in every iteration by taking the average of the forward and backward finite difference approximations.

$$y_{n+1} = y_n + \frac{1}{2}h(f(t_{n+1}, y_{n+1}) \tag{6}$$

As in the previous case, 6 is an implicit equation and thus it might require the use of Newton's method to find a solution.

Figure 1 shows the solution from $t = 0$ to $t = 10$ of the two initial value problems given by explicit, implicit Euler and trapezoidal, along with the true solution. The solution is approximated in both cases by using 30 points ($h = 0.345$).

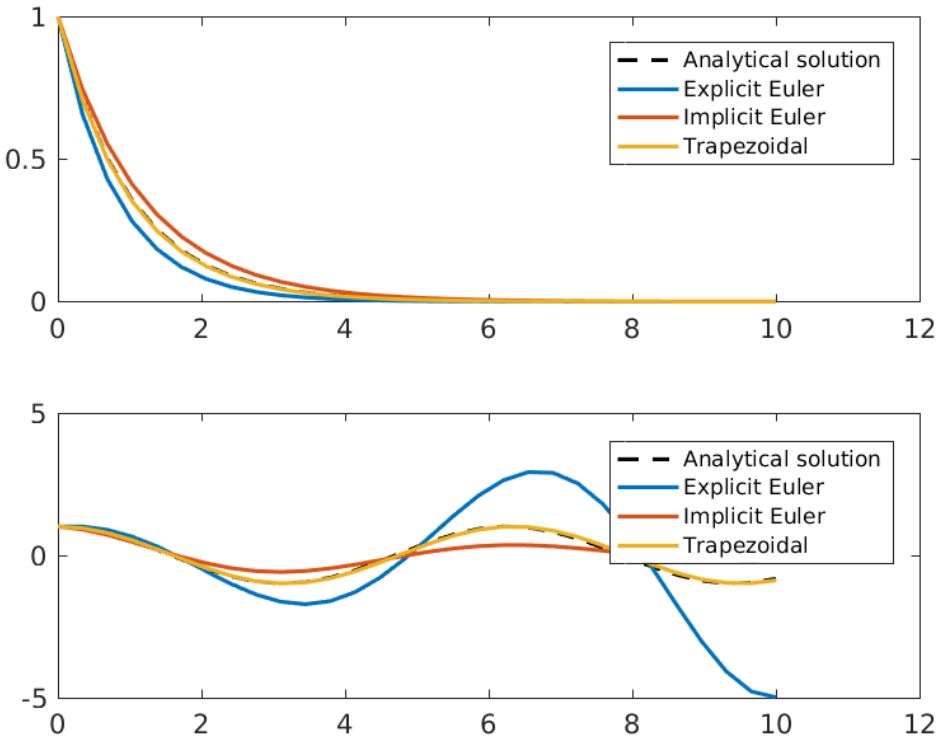

Figure 1: Explicit, Implicit Euler and Trapezoidal method solution to test equation and harmonic oscillator for h = 0.345

All three implimentations of these one step algorithms along with the code for Newton's method can be found in the appendix.

1.2 Multistep methods

Even though we see in figure 1 that the solution found by the trapezoidal method lies very close to the analytical solution, for some applications one could require higher order approximations to obtain more accurate results.

The classical Runge-Kutta method and its higher order variations play with the concept introduced in last section to create finer approximations to the solution. Concretetly the classical Runge-Kutta method takes four stages to compute a weighted avarage of four different slope estimates. The slope estimation at the fourth stage will be based in the information obtained in the three previous stages.

$$
\begin{aligned}
T_1 &= t_n & X_1 &= x_n \\
T_2 &= t_n + \frac{1}{2}h & X_2 &= x_n + \frac{1}{2}hf(T \\
T_3 &= t_n + \frac{1}{2}h & X_3 &= x_n + \frac{1}{2}hf(T \\
T_4 &= t_n + h & X_4 &= x_n + hf(T_3, \\
t_{n+1} &= t_n + h & & \\
x_{n+1} &= x_n + h(\frac{1}{6}f(T_1, X_1) + \frac{1}{2}f(T_2, X_2) + \frac{1}{2}f(T_3, X_3) + \frac{1}{6}f(T_4, X_4)) & &
\end{aligned}
\tag{7}
$$

Equation 7 reveals that since this is an explicit method there is no need for a numerical solver. The coefficients and weights of these equations are often collected in the Butcher's Tableu. This is described more deeply in section 3.

Two different variations of DOPRI54 have also been implemented. DOPRI54 is just another explicit Runge-Kutta method of higher order with more stages and a larger Butcher's Tableu. We shall see when comparing the error estimates the difference between the two methods. The three methods are shown in the appendix.

1.3 Global and local errors

It is easy to see in figure 1, especially in the bottom graph, 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.

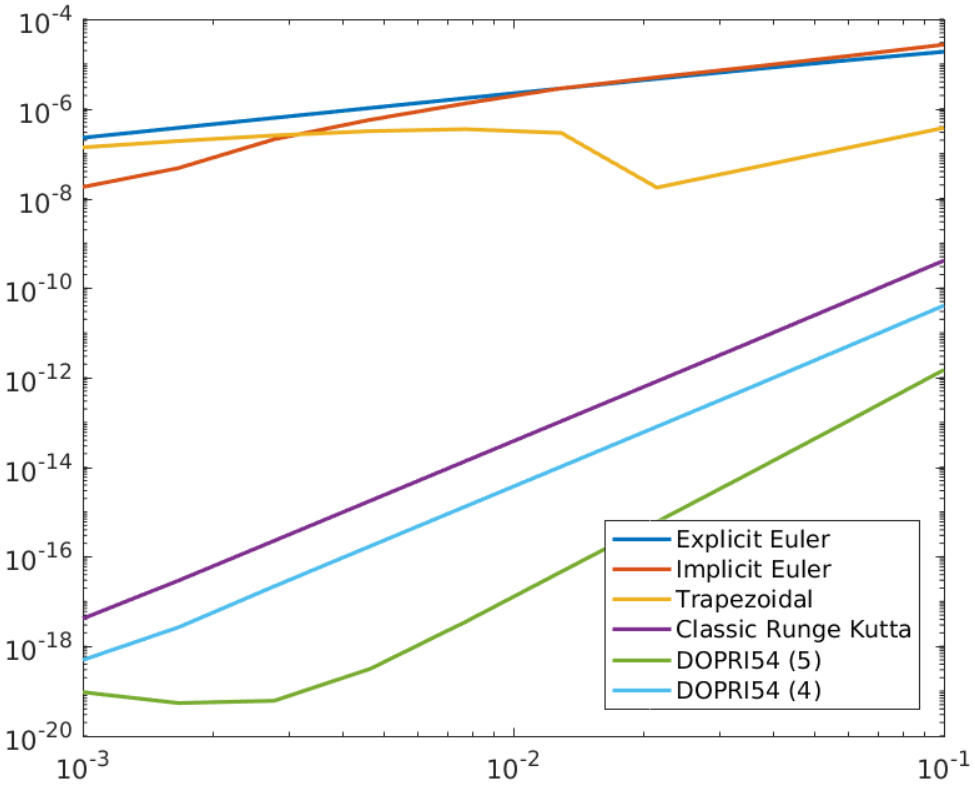

Figure 2: Explicit, Implicit Euler and Trapezoidal method global errors in test equation

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

$$x(t) = \exp(-t) \qquad x(t) = cos(t) \tag{8}$$

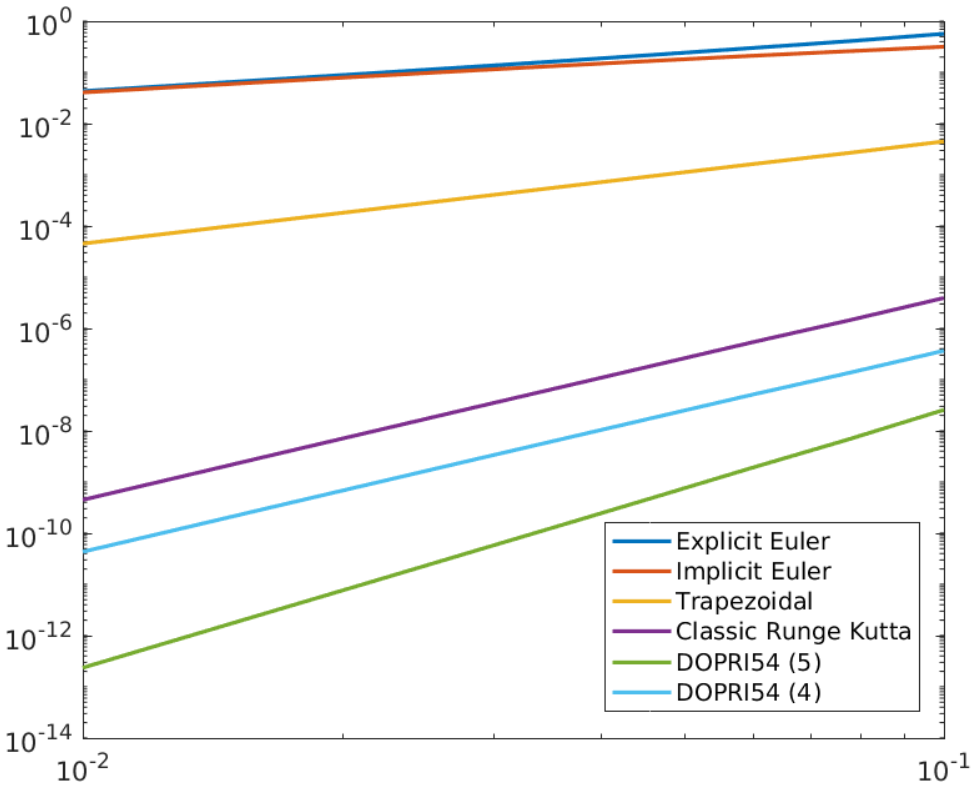

Figure 3: Explicit, Implicit Euler and Trapezoidal method global errors in harmonic oscillator

Figure 2 represents the global error at time $t = 10$ made by the one step solvers (left graph) and the multistep solver (right graph) for different step sizes. As expected, the size the global error decreases when increasing the number of points used in the approximation.

Why is not good to use global error??

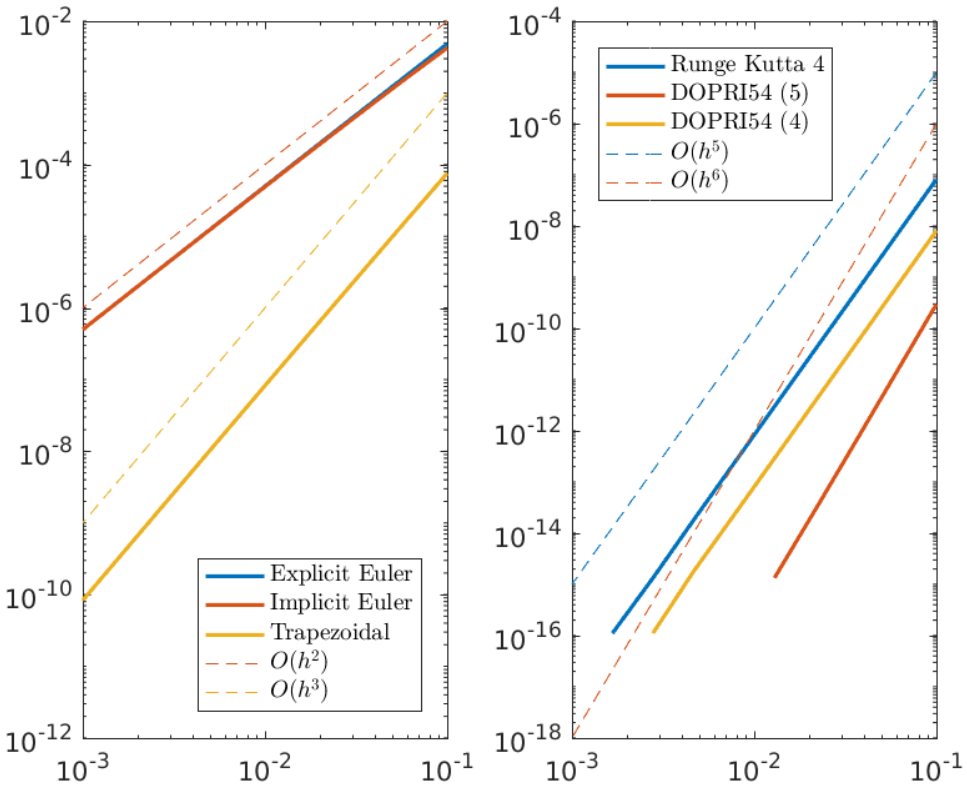

Figure 4: Explicit, Implicit Euler and Trapezoidal method local errors in test equation

On the other hand the local error at time $t = t_0 + h$ is shown in figure 4. 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 verify the order of the solvers. That is, order 1 for Explicit and Implicit Euler, order 2 for the Trapezoidal method, order 4 for the classical Runge Kutta and one of the DOPRI54 versions and order 5 for the other DOPRI54.

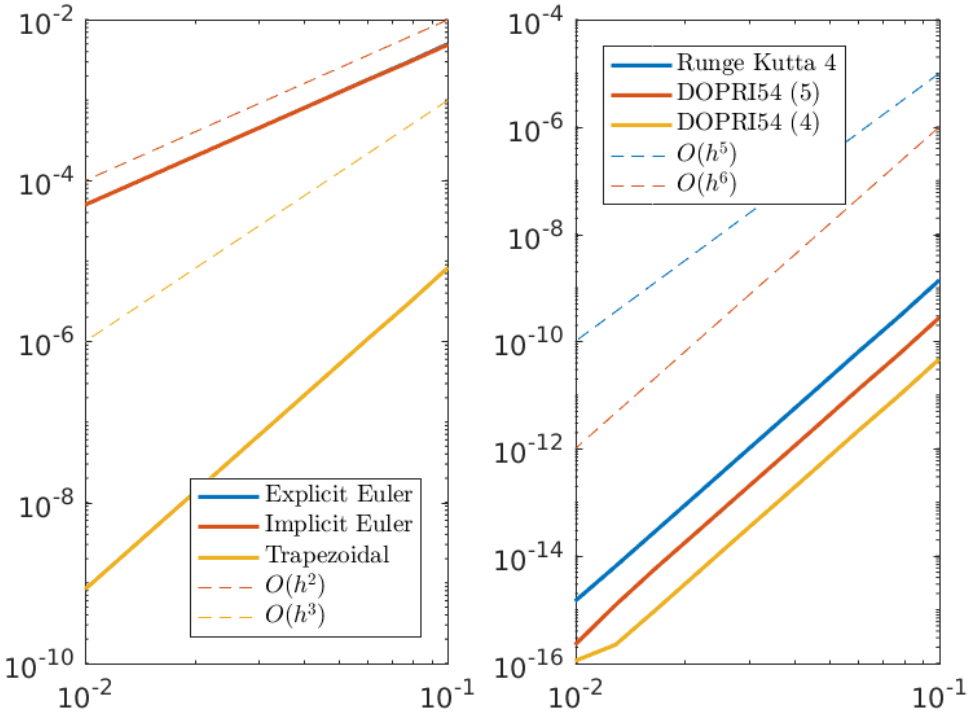

Figure 5: Explicit, Implicit Euler and Trapezoidal method local errors in harmonic oscillator

1.4 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, such as DOPRI54, use embbeded methods of lower order to estimate the error. Even though the estimations obtained as not as good as with step doubling, the secondary method is closely related to the main algorithm so that they can share computations and thus be very efficient.

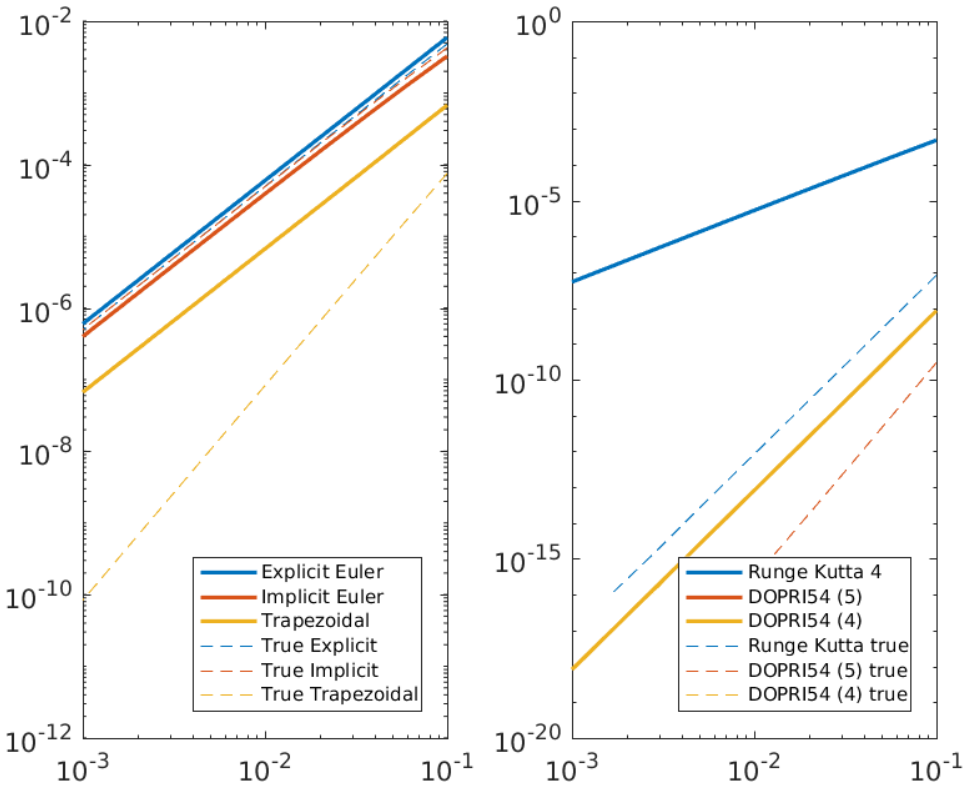

Figure 6: Explicit, Implicit Euler and Trapezoidal method local error estimates in test equation

The local error estimates are plotted along with the true errors in figure 6 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. In the embedded method estimation the slope is related to the order of lower order algorithm.

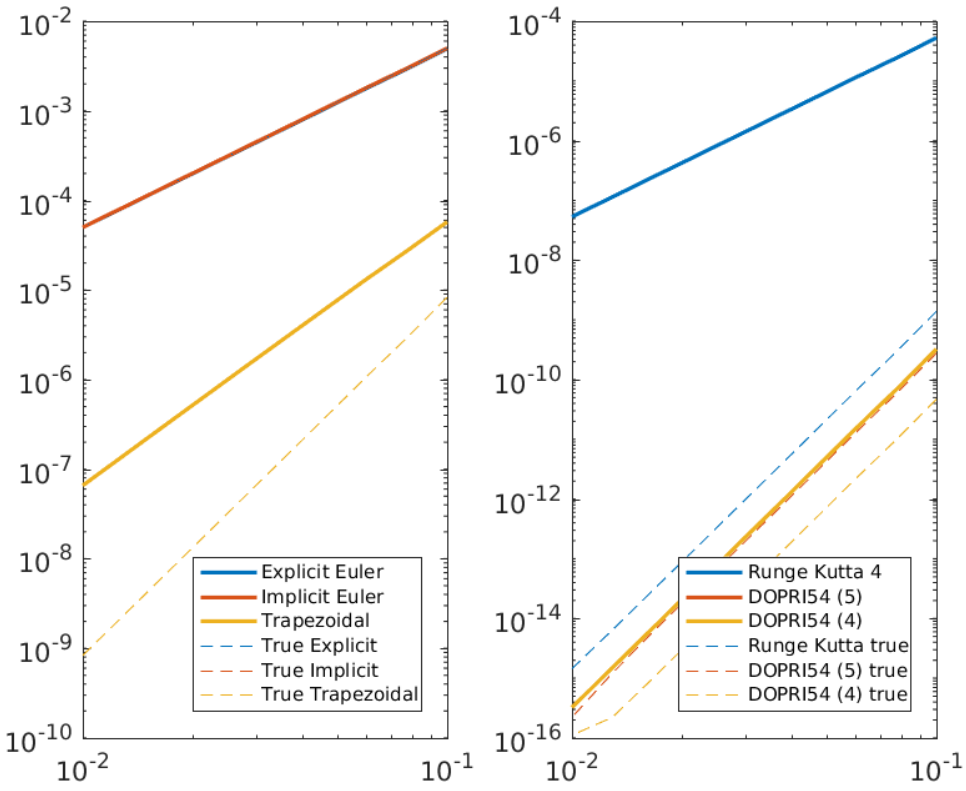

Figure 7: Explicit, Implicit Euler and Trapezoidal method local error estimates in harmonic oscillator

2 The Van der Pol System

In this section we are going to test the implemented methods using the Van der Pol oscillator. This particular initial value problem is very useful to evaluate how numerical solvers respond to stiffness.

Intuituively one can think of stiff problems as those whose solution varies rapidly in a short time span. This characteristic has an strong impact in some numerical methods that might need to take very small steps to get an accurate solution. Since their approximation is based in previous points, explicit methods are specially affected by this phenomena. For these methods, spontaneous variations have an strong effect on the truncation errors and make the numerical solution diverge from the real one. The concept of stiffness is strongly related to the stability of numerical algorithms which is discussed in sections 3 and 5.

Figure 8 shows the solution obtained with the 5 different methods when $\mu = 3$. The graph on the left is the error estimation. In this case, all methods are able to converge to the true solution without having to take very small steps.

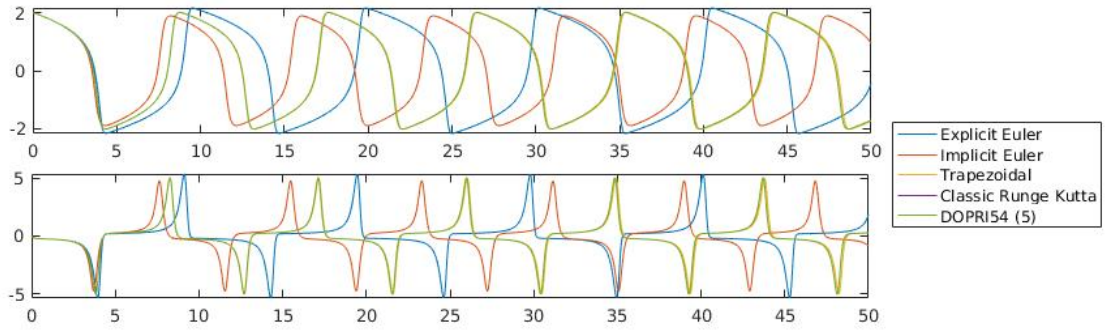

Figure 8: Van der Pol oscillator solution from $t = 0$ to $t = 50$ for $\mu = 3$ and $h = 0.05$

On the other hand, figure 9 shows the solution for $\mu = 100$. For this value of the parameter the problem becomes very stiff in some regions. The bottom graph represents the degree of variation and the peaks give a measure of how fast the solution moves. We see that while the explicit algorithms are unable to handle the problem and their error blows up, implicit methods such as backward Euler can easily follow the solution track because the rely on future values and thus, they are aware in advance of those rapid variations.

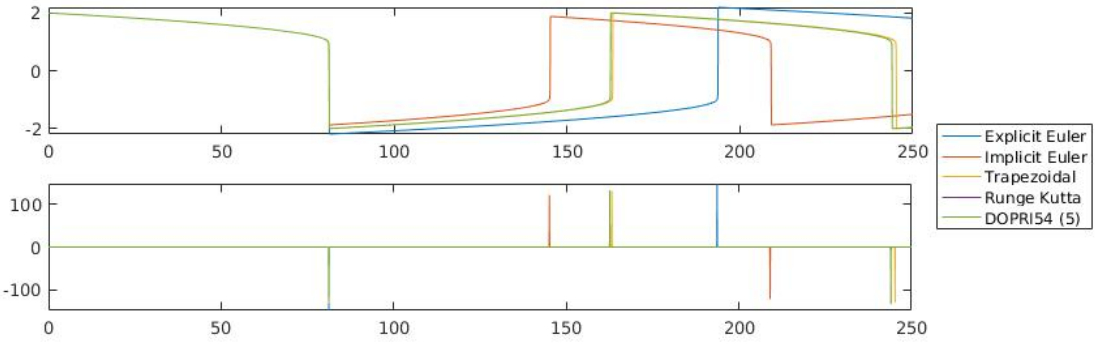

Figure 9: Van der Pol oscillator solution from $t = 0$ to $t = 250$ for $\mu = 100$ and $h = 0.0025$

We have measured the local error at $t = t_0 + h$ for different step sizes and $\mu = 100$. Figure 10 shows that the implicit methods, that is, Implicit Euler and

trapezoidal are accurate even when taking large steps. On the other hand, the error of the high order explicit methods becomes extremely large for step sizes of 10^-1. The fact that the trapezoidal method is an hybrid of the forward and the backward finit difference approximations explains why implicit Euler perfoms better for large steps.

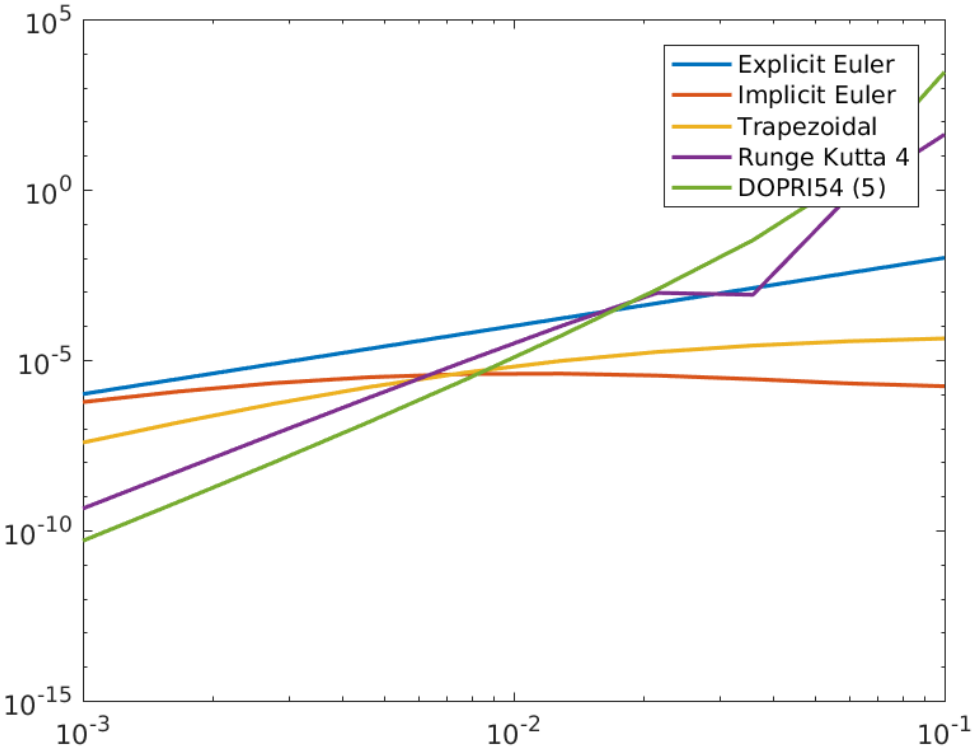

Figure 10: Van der Pol oscillator local error estimates for $\mu = 100$

3 Design your own Explicit Runge-Kutta Method

3.1 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:

$$\mathcal{O}(h^1): \quad b^T e = 1 \quad b_1 + b_2 + b_3 = 1 \quad \tau_1 \to \bullet \tag{9a}$$

$$\mathcal{O}(h^2): \quad b^T C e = \frac{1}{2} \quad \underbrace{b_1 c_1}_{0} + b_2 c_2 + b_3 c_3 = \frac{1}{2} \quad \tau_2 \to \vdots \tag{9b}$$

$$\mathcal{O}(h^3): \quad b^T C^2 e = \frac{1}{3} \quad \underbrace{b_1 c_1^2}_{0} + b_2 c_2^2 + b_3 c_3^2 = \frac{1}{3} \quad \tau_3 \to \therefore \tag{9c}$$

$$b^T A C e = \frac{1}{6} \quad \underbrace{b_2 a_{21} c_1}_{0} + \underbrace{b_3 a_{31} c_1}_{0} + b_3 a_{32} c_2 = \frac{1}{6} \quad \tau_4 \to \vdots \tag{9d}$$

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.

$$c_2 = a_{21} \tag{9e}$$

$$c_3 = a_{31} + a_{32} \tag{9f}$$

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].

$$\widehat{b}_1 + \widehat{b}_2 + \widehat{b}_3 = 1 \tag{10a}$$

$$\widehat{b}_2 c_2 + \widehat{b}_3 c_3 = \frac{1}{2} \tag{10b}$$

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

3.2 Testing on the test equation

Figure 11 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.

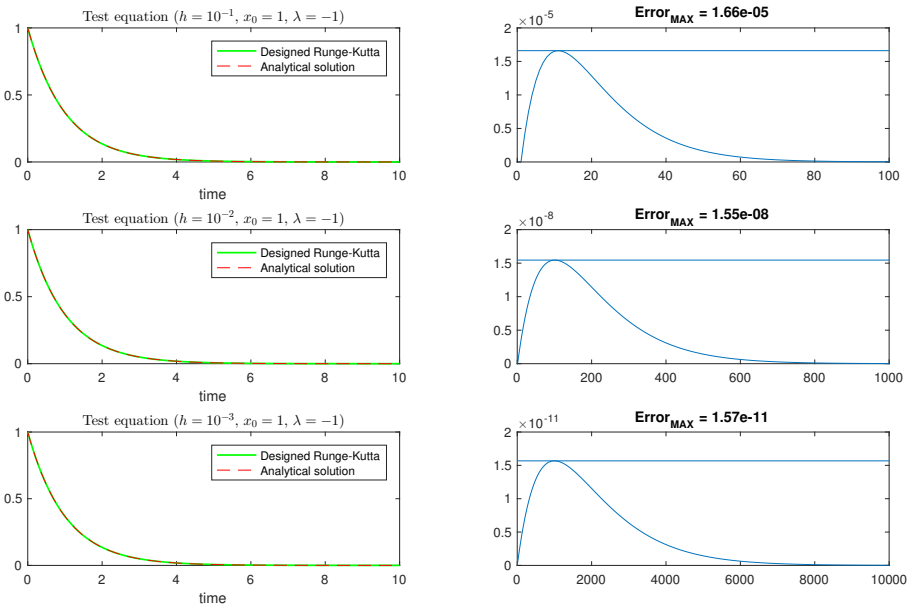

Figure 11: Comparison with the test equation for different step sizes

3.3 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 12 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.

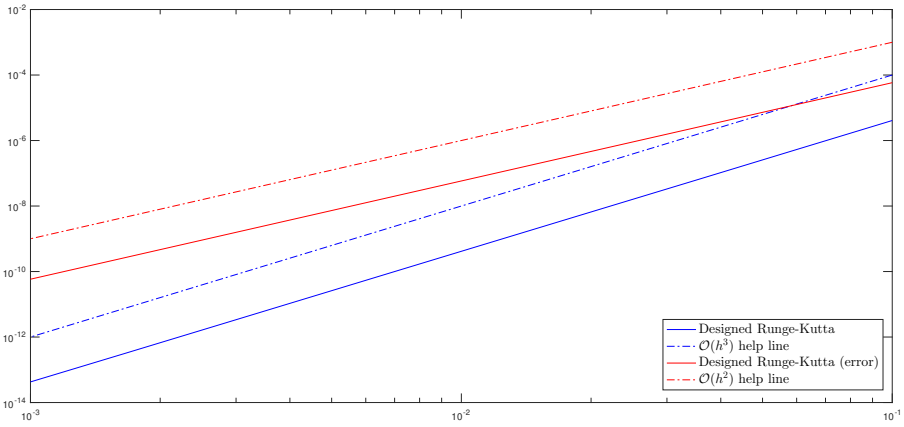

Figure 12: Loglog plot of the local error of designed ERK method (blue) and the returned error of the method (red, local error estimate) with help lines

3.4 $\mathbf{R}(\lambda \mathrm{h})$ 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 13. Although the plots look similar, the embedded method's stability region is slightly smaller, as well as all other metrics.

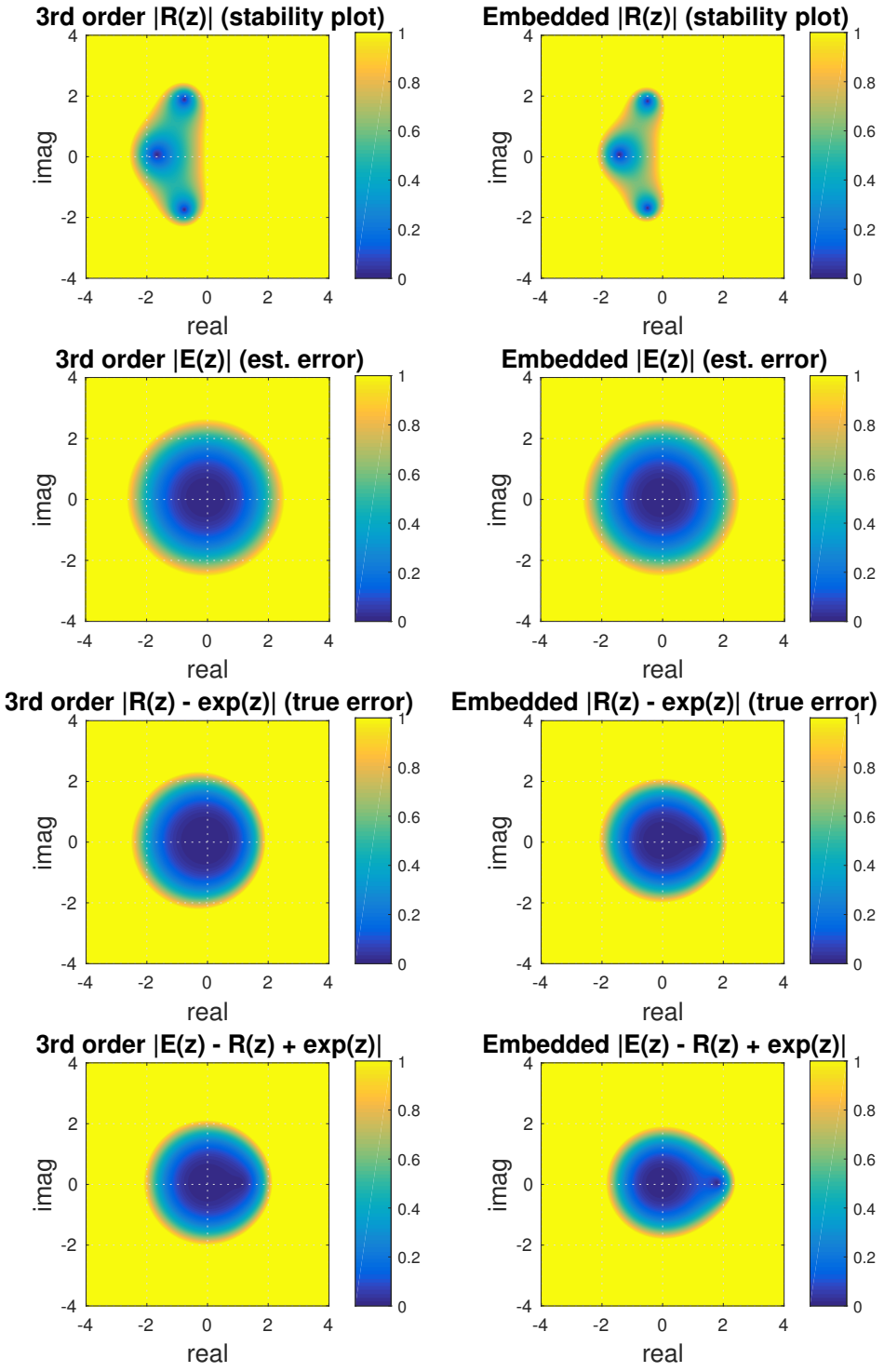

Figure 13: Stability plots of the third order ERK with second order embedded method. In order for the method to be A stable the whole left half plane has to be $|R(z)| < 1$ (i.e. in graphical representation shown not brightly yellow) which clearly it is not.

3.5 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.

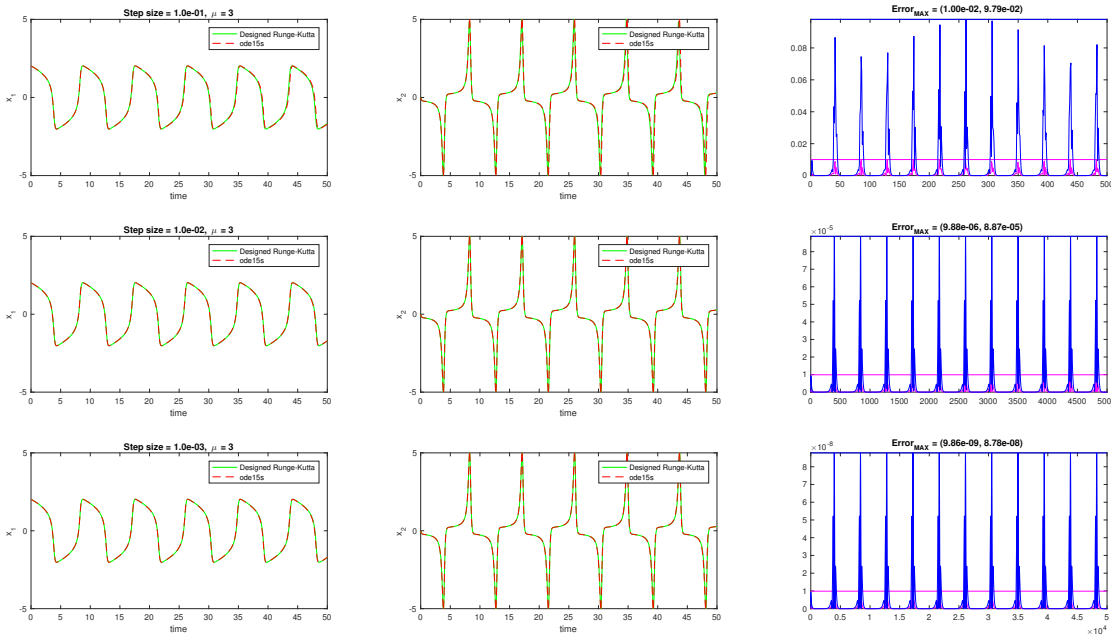

Figure 14: Comparison with `ode15s` on Van der Pol problem ($\mu = 3$). Each row depicts different step size (0.1, 0.01 and 0.001) and the maximal error from ERK is shown in the plot title as well as on a reference line (x_1 - magenta, x_2 - blue).

4 Step size controller

Ideally we would like to be able to obtain an accurate solution without having to take too many unnecesary steps. With this intention in mind we are going to modify the code, so that instead of having a fixed step size, we will give our method the ability to increase or decrease it depending on the error estimation.

Since we already know how to estimate the local error (step doubling or embbeded methods) we can check whether the error made for a specific step size is

Absolute tol $= 10^{-3}$ Relative tol $= 10^{-3}$			
Method	Evaluations	Accepted	Rejected
Explicit Euler	3175	1507	163
Implicit Euler	21528	1490	186
Trapezoidal	18959	504	199
Runge-Kutta 4	10949	650	381
DROPI54	1749	178	85

Table 3: Number of function evaluations number of steps accepted and number of steps rejected using an asymptotic controller

Absolute tol $= 10^{-5}$ Relative tol $= 10^{-5}$			
Method	Evaluations	Accepted	Rejected
Explicit Euler	29847	14922	5
Implicit Euler	139588	14860	5
Trapezoidal	47931	2190	133
Runge-Kutta 4	67155	6056	55
DROPI54	3487	380	139

Table 4: Number of function evaluations number of steps accepted and number of steps rejected using an asymptotic controller

below a given tolerance. In case the error is larger than this tolerance, we reject the step and we update its size.

There are many ways one can control and update the step size. In our case we will make use of asymptotic and PI controllers. The former makes the update taking into account just the current value of the error, while the latter also relies on the previous estimation.

We have tested the five different methods for different values of the absolute and relative tolerances. The results using an asymptotic controller with relative and absolute tolerance of 10^{-3} are gathered in table **??**.

We see in table **??** that the number of function evaluations for the implicit methods is usually high. The reason is that this methods require a numerical solver, in this case Newton's method. It is also important to notice that, since DROPI54 uses an embedded method to estimate the error, the number of function evaluations is lower than for the rest of the methods, which compute those estimates by step doubling.

We have also collected the results obtained for both asymptotic and PI controller with absolute and relative tolerances of 10^{-5} in tables **??** and **??**. The results show that there is not much difference between the two updating methods

Absolute tol $= 10^{-5}$ Relative tol $= 10^{-5}$			
Method	Evaluations	Accepted	Rejected
Explicit Euler	29856	14922	14
Implicit Euler	139674	14860	15
Trapezoidal	54563	2202	443
Runge-Kutta 4	69431	6062	276
DROPI54	5734	383	510

Table 5: Number of function evaluations number of steps accepted and number of steps rejected using a PI controller

Absolute tol $= 10^{-8}$ Relative tol $= 10^{-8}$			
Method	Evaluations	Accepted	Rejected
Explicit Euler	943470	471728	17
Implicit Euler	4621862	730992	65
Trapezoidal	350341	24438	367
Runge-Kutta 4	2118392	192553	32
DROPI54	13772	1389	676

Table 6: Number of function evaluations number of steps accepted and number of steps rejected using a PI controller

apart from the fact that PI takes a few more steps than the asymptotic controller.

Table **??** shows the values obtained for absolute and relative tolerances of 10^-8 with a PI controller. In this case the number of steps required for all methods except DOPRI54 is extremely large. That is why for such small tolerances high order methods are useful in order to prevent long run time and increase the computing performance.

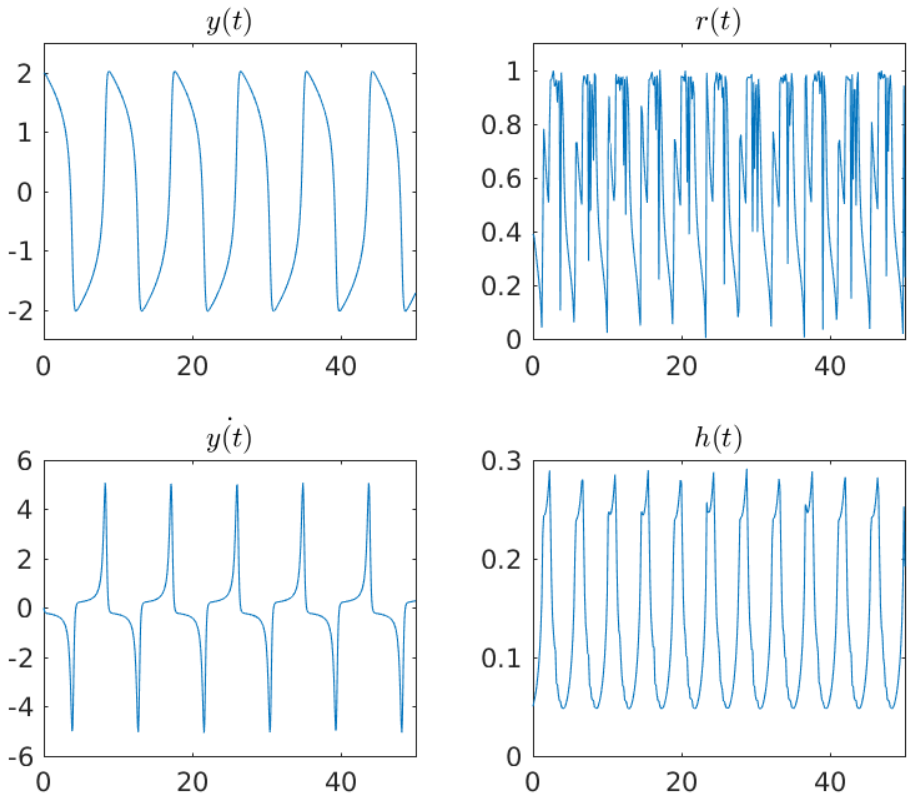

Figure 15: Solution of the Van der Pol oscillator for DOPRI 54 with adaptive step size.

Finally figure 15 shows the solution found with DOPRI54, along with the step sizes chosen at every point in time. As expected, the closer the points are to the drops shown in the top left graph the more steps are needed to obtain an accurate solution and the shorter those steps are.

5 ESDIRK23

5.1 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.

5.2 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 16 (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.

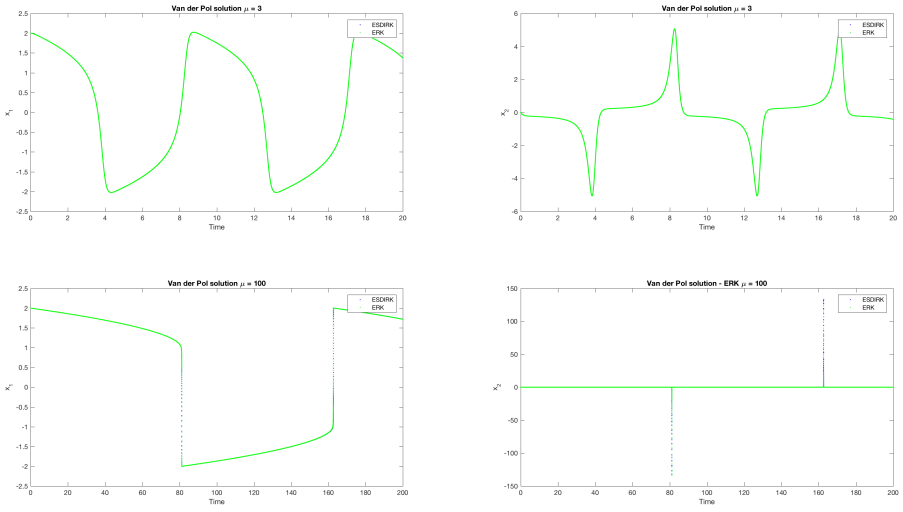

Figure 16: ESDIRK23 vs our ERK method on the Van der Pol problem with $\mu = 3$ and $\mu = 100$ (stiff)

5.3 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 cal-

culate 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.

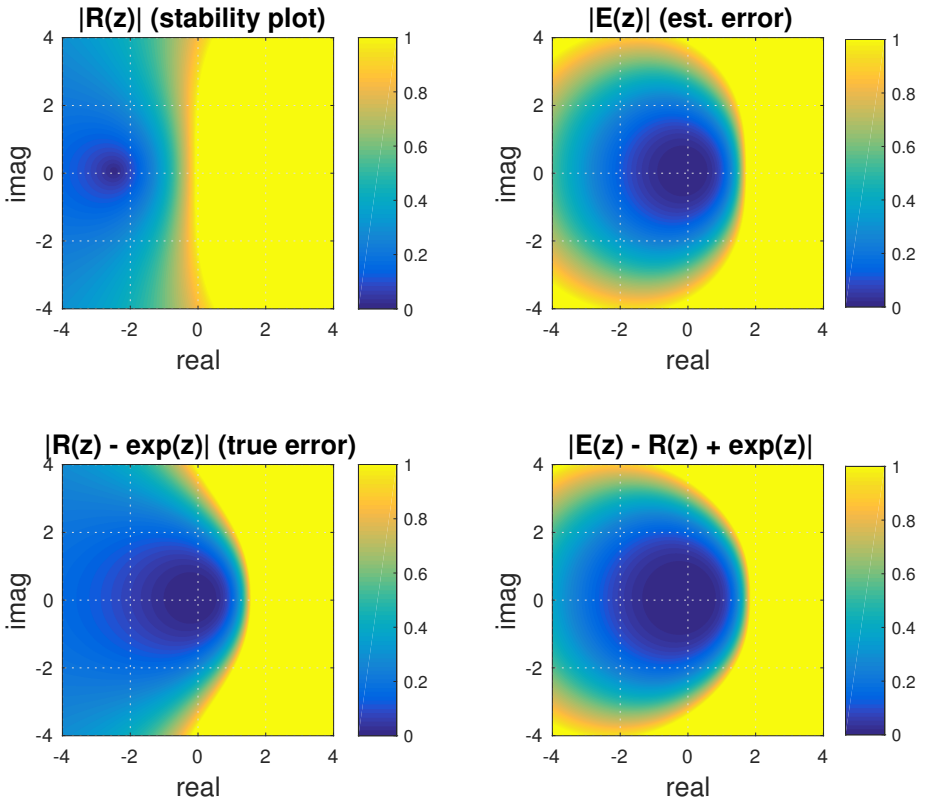

Figure 17: ESDIRK23 stability summary

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 ESDIRK23 or the trapezoidal method.

5.4 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

```
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
```

```
41              'nFun',       0, ...
42              'nJac',       0, ...
43              'nLU',        0, ...
44              'nBack',      0, ...
45              'nStep',      0, ...
46              'nAccept',    0, ...
47              'nFail',      0, ...
48              'nDiverge',   0, ...
49              'nSlowConv', 0);
50
51
52
53  % Main ESDIRK Integrator
54  %
        ================================================================================

55  nx = size(x0,1);
56  F = zeros(nx,s);
57  t = t0;
58  x = x0;
59  h = h0;
60  IG = eye(length(x0)); % ex5 replacement for g
61
62  [F(:,1),~]  = feval(fun,t,x,varargin{:}); % ex5 no need for g
63  info.nFun = info.nFun+1;
64  [dfdx,~] = feval(jac,t,x,varargin{:}); % ex5 no need for g
65  info.nJac = info.nJac+1;
66  %FreshJacobian = true; % ex5
67  if (t+h)>tf
68      h = tf-t;
69  end
70  hgamma = h*gamma;
71  dRdx = IG - hgamma*dfdx;
72  [L,U,pivot] = lu(dRdx,'vector');
73  info.nLU = info.nLU+1;
74  %hLU = h; % ex5
75
76  %FirstStep = true; % ex5
77  %ConvergenceRestriction = false; % ex5
78  %PreviousReject = false; % ex5
79  iter = zeros(1,s);
80
81  % Output
82  chunk = 100;
83  Tout = zeros(chunk,1);
84  Xout = zeros(chunk,nx);
85  %Gout = zeros(chunk,nx); % ex5
86
87  Tout(1,1) = t;
```

```
88  Xout(1,:) = x.';
89  %Gout(1,:) = g.'; % ex5
90
91  while t<tf
92      info.nStep = info.nStep+1;
93      %
            ========================================================================

94      % A step in the ESDIRK method
95      i=1;
96      diverging = false;
97      SlowConvergence = false; % carsten
98      alpha = 0.0;
99      Converged = true;
100     while (i<s) && Converged
101         % Stage i=2,...,s of the ESDIRK Method
102         i=i+1;
103         phi = x + F(:,1:i-1)*(h*AT(1:i-1,i)); % ex5
104
105         % Initial guess for the state % ex5 removed duplicate code
106         dt = c(i)*h;
107         %G = g + dt*F(:,1);
108         X = x + dt*F(:,1); % ex5
109         T = t+dt;
110
111         [F(:,i),~] = feval(fun,T,X,varargin{:}); % ex5
112         info.nFun = info.nFun+1;
113         R = X - hgamma*F(:,i) - phi; % ex5
114         rNewton = norm(R./(absTol + abs(X).*relTol), inf); % ex5
115         Converged = (rNewton < tau);
116
117         % Newton Iterations
118         while ~Converged && ~diverging && ~SlowConvergence
119             iter(i) = iter(i)+1;
120             dX = U\(L\(R(pivot,1)));
121             info.nBack = info.nBack+1;
122             X = X - dX;
123             rNewtonOld = rNewton;
124             [F(:,i),~] = feval(fun,T,X,varargin{:}); % ex5
125             info.nFun = info.nFun+1;
126             R = X - hgamma*F(:,i) - phi; % ex5
127             rNewton = norm(R./(absTol + abs(X).*relTol), inf); % ex5
128             alpha = max(alpha,rNewton/rNewtonOld);
129             Converged = (rNewton < tau);
130             diverging = (alpha >= 1);
131             SlowConvergence = (iter(i) >= itermax);
132         end
133         diverging = (alpha >= 1)*i; % carsten, recording which stage
                is diverging
```

```
134     end
135     %if diverging, i, iter, pause, end
136     nstep = info.nStep;
137     stats.t(nstep) = t;
138     stats.h(nstep) = h;
139     stats.r(nstep) = NaN;
140     stats.iter(nstep,:) = iter;
141     stats.Converged(nstep) = Converged;
142     stats.Diverged(nstep)  = diverging;
143     stats.AcceptStep(nstep) = false;
144     stats.SlowConv(nstep)  = SlowConvergence*i; % carsten, recording
            which stage is converging to slow (reaching maximum no. of
            iterations)
145     iter(:) = 0; % carsten
146     %
            ========================================================================

147
148     % Error estimation
149     e = F*(h*d);
150     r = norm(e./(absTol + abs(X).*relTol), inf); % ex5
151     r = max(r,eps);
152     stats.r(nstep) = r;
153     t = T;
154     x = X;
155     F(:,1) = F(:,s);
156
157     % Jacobian Update Strategy
158     [dfdx,~] = feval(jac,t,x,varargin{:}); % ex5
159     info.nJac = info.nJac+1;
160     hgamma = h*gamma;
161     dRdx = IG - hgamma*dfdx; % ex5
162     [L,U,pivot] = lu(dRdx,'vector');
163     info.nLU = info.nLU+1;
164     info.nFail = info.nFail + ~Converged; % ex5
165     info.nDiverge = info.nDiverge + (~Converged && diverging); % ex5
166
167     %
            ========================================================================

168     % Storage of variables for output % ex5
169     info.nAccept = info.nAccept + 1;
170     nAccept = info.nAccept;
171     if nAccept > length(Tout);
172        Tout = [Tout; zeros(chunk,1)];
173        Xout = [Xout; zeros(chunk,nx)];
174     end
175     Tout(nAccept,1) = t;
176     Xout(nAccept,:) = x.';
```

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

B ESDIRK23 variable step size

```
1  function [Tout,Xout,info,stats] = ESDIRK23_Adaptive(fun,jac,t0,tf,x0,
       h0,absTol,relTol,varargin)
2
3  % ESDIRK23 Adaptive
4  % Modified for Ex5
5  %
       ================================================================================

6  % Runge-Kutta method parameters
7  gamma = 1-1/sqrt(2);
8  a31 = (1-gamma)/2;
9  AT = [0 gamma a31;0 gamma a31;0 0 gamma];
10 c  = [0; 2*gamma; 1];
11 b  = AT(:,3);
12 bhat = [    (6*gamma-1)/(12*gamma); ...
13             1/(12*gamma*(1-2*gamma)); ...
14             (1-3*gamma)/(3*(1-2*gamma))    ];
15 d  = b-bhat;
16 p  = 2;
17 phat = 3;
18 s = 3;
19
20
21 % error and convergence controller
22 epsilon = 0.8;
23 tau = 0.1*epsilon; %0.005*epsilon;
24 itermax = 20;
25 ke0 = 1.0/phat;
26 ke1 = 1.0/phat;
27 ke2 = 1.0/phat;
28 alpharef = 0.3;
29 alphaJac = -0.2;
30 alphaLU  = -0.2;
31 hrmin = 0.01;
32 hrmax = 10;
33 %
       ================================================================================

34 tspan = [t0 tf]; % carsten
35 info = struct(...
36             '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);
```

```
84  Xout = zeros(chunk,nx);
85
86  Tout(1,1) = t;
87  Xout(1,:) = x.';
88
89  while t<tf
90      info.nStep = info.nStep+1;
91      %
            ==========================================================================

92      % A step in the ESDIRK method
93      i=1;
94      diverging = false;
95      SlowConvergence = false; % carsten
96      alpha = 0.0;
97      Converged = true;
98      while (i<s) && Converged
99          % Stage i=2,...,s of the ESDIRK Method
100         i=i+1;
101         phi = x + F(:,1:i-1)*(h*AT(1:i-1,i)); % ex5
102
103         % Initial guess for the state
104         dt = c(i)*h;
105         X  = x + dt*F(:,1); % ex5
106         T = t+dt;
107
108         [F(:,i),~] = feval(fun,T,X,varargin{:}); % ex5
109         info.nFun = info.nFun+1;
110         R = X - hgamma*F(:,i) - phi; % ex5
111         rNewton = norm(R./(absTol + abs(X).*relTol), inf);
112         Converged = (rNewton < tau);
113
114         % Newton Iterations
115         while ~Converged && ~diverging && ~SlowConvergence
116             iter(i) = iter(i)+1;
117             dX = U\(L\(R(pivot,1)));
118             info.nBack = info.nBack+1;
119             X = X - dX;
120             rNewtonOld = rNewton;
121             [F(:,i),~] = feval(fun,T,X,varargin{:}); % ex5
122             info.nFun = info.nFun+1;
123             R = X - hgamma*F(:,i) - phi; % ex5
124             rNewton = norm(R./(absTol + abs(X).*relTol), inf);
125             alpha = max(alpha,rNewton/rNewtonOld);
126             Converged = (rNewton < tau);
127             diverging = (alpha >= 1);
128             SlowConvergence = (iter(i) >= itermax); % carsten
129         end
130         diverging = (alpha >= 1)*i; % carsten, recording which stage
```

```
            is diverging
131     end
132
133     nstep = info.nStep;
134     stats.t(nstep) = t;
135     stats.h(nstep) = h;
136     stats.r(nstep) = NaN;
137     stats.iter(nstep,:) = iter;
138     stats.Converged(nstep) = Converged;
139     stats.Diverged(nstep)  = diverging;
140     stats.AcceptStep(nstep) = false;
141     stats.SlowConv(nstep)  = SlowConvergence*i; % carsten, recording
            which stage is converging to slow (reaching maximum no. of
            iterations)
142     iter(:) = 0; % carsten
143     %
            ========================================================================

144     % Error and Convergence Controller
145     if Converged
146         % Error estimation
147         e = F*(h*d);
148         r = norm(e./(absTol + abs(X).*relTol), inf);
149         CurrentStepAccept = (r<=1.0);
150         r = max(r,eps);
151         stats.r(nstep) = r;
152         % Step Length Controller
153         if CurrentStepAccept
154             stats.AcceptStep(nstep) = true;
155             info.nAccept = info.nAccept+1;
156             if FirstStep || PreviousReject || ConvergenceRestriction
157                 % Aymptotic step length controller
158                 hr = 0.75*(epsilon/r)^ke0;
159             else
160                 % Predictive controller
161                 s0 = (h/hacc);
162                 s1 = max(hrmin,min(hrmax,(racc/r)^ke1));
163                 s2 = max(hrmin,min(hrmax,(epsilon/r)^ke2));
164                 hr = 0.95*s0*s1*s2;
165             end
166             racc = r;
167             hacc = h;
168             FirstStep = false;
169             PreviousReject = false;
170             ConvergenceRestriction = false;
171
172             % Next Step
173             t = T;
174             x = X;
```

```
175             F(:,1) = F(:,s);
176
177         else % Reject current step
178             info.nFail = info.nFail+1;
179             if PreviousReject
180                 kest = log(r/rrej)/(log(h/hrej));
181                 kest = min(max(0.1,kest),phat);
182                 hr   = max(hrmin,min(hrmax,((epsilon/r)^(1/kest))));
183             else
184                 hr = max(hrmin,min(hrmax,((epsilon/r)^ke0)));
185             end
186             rrej = r;
187             hrej = h;
188             PreviousReject = true;
189         end
190
191         % Convergence control
192         halpha = (alpharef/alpha);
193         if (alpha > alpharef)
194             ConvergenceRestriction = true;
195             if hr < halpha
196                 h = max(hrmin,min(hrmax,hr))*h;
197             else
198                 h = max(hrmin,min(hrmax,halpha))*h;
199             end
200         else
201             h = max(hrmin,min(hrmax,hr))*h;
202         end
203         h = max(1e-8,h);
204         if (t+h) > tf
205             h = tf-t;
206         end
207
208         % Jacobian Update Strategy
209         FreshJacobian = false;
210         if alpha > alphaJac
211             [dfdx,~] = feval(jac,t,x,varargin{:}); % ex5
212             info.nJac = info.nJac+1;
213             FreshJacobian = true;
214             hgamma = h*gamma;
215             dRdx = IG - hgamma*dfdx;  % ex5
216             [L,U,pivot] = lu(dRdx,'vector');
217             info.nLU = info.nLU+1;
218             hLU = h;
219         elseif (abs(h-hLU)/hLU) > alphaLU
220             hgamma = h*gamma;
221             dRdx = IG - hgamma*dfdx; % ex5
222             [L,U,pivot] = lu(dRdx,'vector');
223             info.nLU = info.nLU+1;
```

```
224             hLU = h;
225         end
226     else % not converged
227         info.nFail=info.nFail+1;
228         CurrentStepAccept = false;
229         ConvergenceRestriction = true;
230         if FreshJacobian && diverging
231             h = max(0.5*hrmin,alpharef/alpha)*h;
232             info.nDiverge = info.nDiverge+1;
233         elseif FreshJacobian
234             if alpha > alpharef
235                 h = max(0.5*hrmin,alpharef/alpha)*h;
236             else
237                 h = 0.5*h;
238             end
239         end
240         if ~FreshJacobian
241             [dfdx,~] = feval(jac,t,x,varargin{:}); % ex5
242             info.nJac = info.nJac+1;
243             FreshJacobian = true;
244         end
245         hgamma = h*gamma;
246         dRdx = IG - hgamma*dfdx; % ex5
247         [L,U,pivot] = lu(dRdx,'vector');
248         info.nLU = info.nLU+1;
249         hLU = h;
250     end
251
252     %
        ==========================================================================

253     % Storage of variables for output
254
255     if CurrentStepAccept
256        nAccept = info.nAccept;
257        if nAccept > length(Tout);
258            Tout = [Tout; zeros(chunk,1)];
259            Xout = [Xout; zeros(chunk,nx)];
260        end
261        Tout(nAccept,1) = t;
262        Xout(nAccept,:) = x.';
263     end
264 end
265 info.nSlowConv = length(find(stats.SlowConv)); % carsten
266 nAccept = info.nAccept;
267 Tout = Tout(1:nAccept,1);
268 Xout = Xout(1:nAccept,:);
```