DANMARKS TEKNISKE UNIVERSITET

02685 ASSIGNMENT 01

Ordinary Differential Equations

Arturo Arranz Mateo (s160412)
Manuel Montoya Catala (s162706)

March 2017

Contents

1 The Test Problem and DOPRI54

1.1 Implement the following methods for these problems: 1) the explicit Euler method, 2) the Implicit Euler method, 3) the Trapezoidal Method, 4) the Classical Runge-Kutta method, 5) the two methods in DOPRI54

In the implementation of these methods we have built 1,2,3 and 4 from scratch, using the Newton's method to resolve the implicit equations of 2 and 3. For the implementation of 5, DOPRI54, we just simplified the solver code given by the professor John Bagterp in the function ERKSolverErrorEstimation.m.

1.2 Derive the analytical solutions for these problems

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

For solving the first IVP,(1), the method of variables separation is used. The detailed steps are shown in (2).

$$\begin{aligned} \dot{x}(t) &= \lambda x(t) \\ \frac{dx}{dt} &= \lambda x(t) \\ \int_{x_0}^{x} \frac{dx}{x} &= \int_{t_0}^{t} \lambda dt \\ ln(x) - ln(x_0) &= \lambda(t - t_0) \\ x &= e^{\lambda(t-t_0)} e^{ln(x_0)} \\ x(t) &= e^{\lambda(t-t_0)} x_0 \\ x(t) &= e^{t} \end{aligned} \tag{2}$$

$$\ddot{x}(t) = -x(t) \quad x(0) = 1 \quad \dot{x}(0) = 0 \tag{3}$$

The IVP (3) can be expressed as a couple of ODEs. Then it can be represented in a matrix form as (5) and (**??**). Then the solution is similar to the scalar case with the exception that in (6) it is represented by the matrix exponential.

$$\dot{X}(t) = AX(t) \tag{4}$$

$$X = \begin{bmatrix} x \\ y \end{bmatrix} \quad A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \quad X(0) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{5}$$

$$X(t) = e^{A*t} X(0) \tag{6}$$

1.3 Plot the global error of the numerical solution at time t = 10.

In Figure 1 we represent the first IVP and all the implemented solvers ($h = 0.25$). It can be observed that it converges to 0 since the exponent has negative real part. In Figure 2 the error convergence is shown. As expected, the higher order methods make better estimation of the solution.

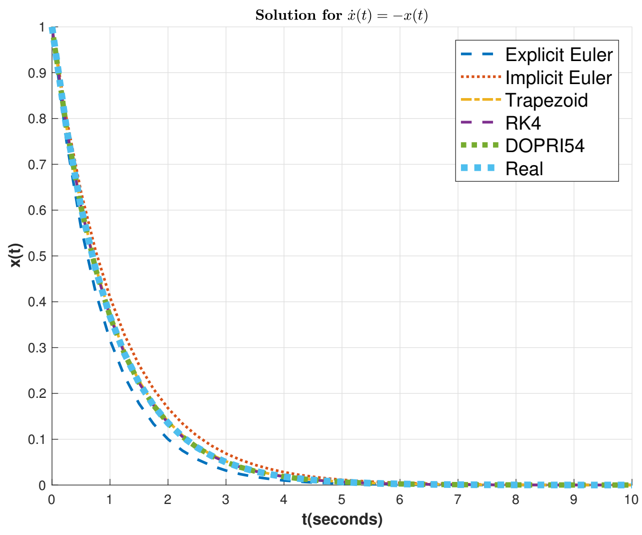
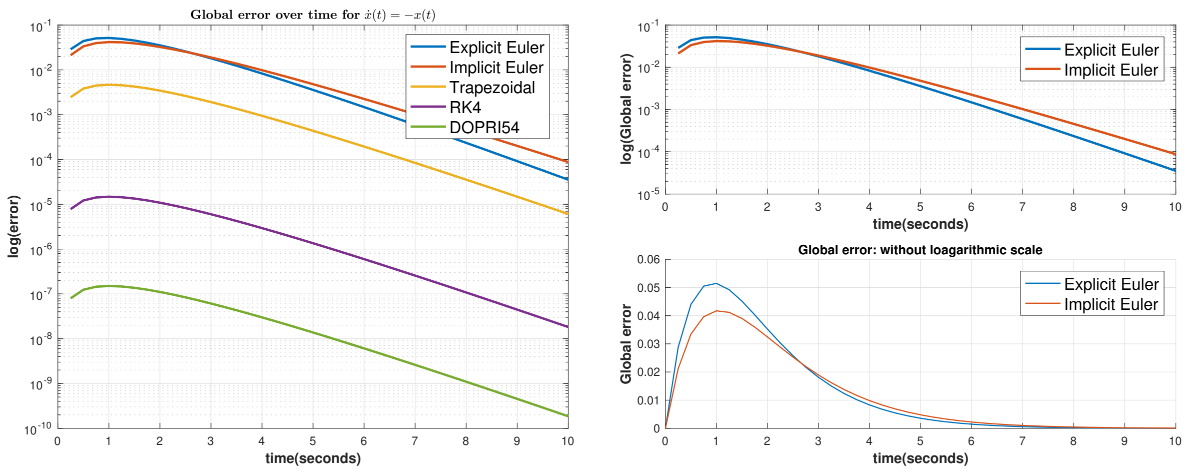


Figure 1: Temporal evolution of IVP1 solutions



(a) All solvers

(b) Euler methods

Figure 2: Global error over time for IVP1: $\dot{x}(t) = -x(t) \quad x(0) = 1$

For the second IVP, the solution has a oscillatory behaviour and does not converge in the span of 10 seconds. Now from figure 3 the solvers solutions difference is much more obvious. All the explicit solvers global error grow exponentially over time. This is represented in the figure **??** where the y-axis is in logarithmic scale. However, the *grow* of the global error for Implicit methods(Euler and Trapezoid) methods seems to decrease over time.

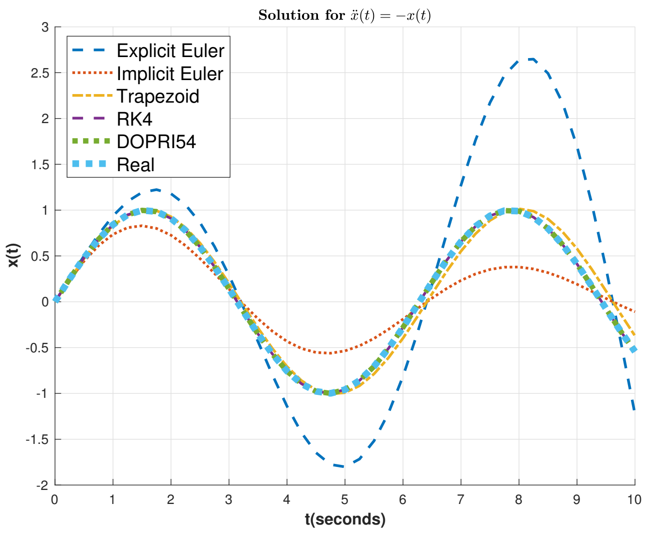
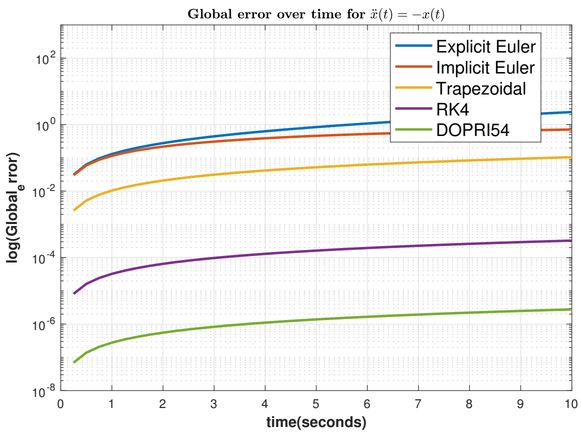
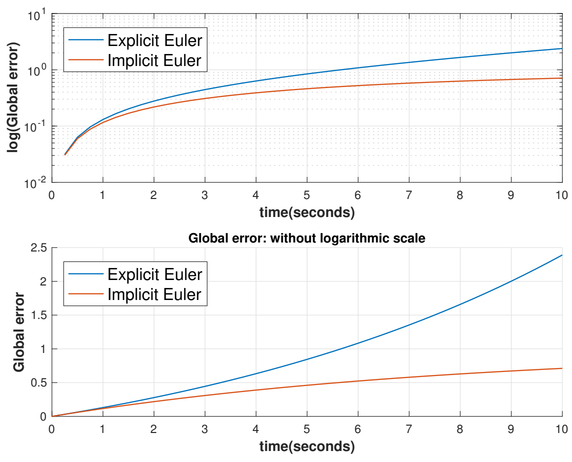


Figure 3: Temporal evolution of IVP2 solutions



(a) All solvers



(b) Euler methods

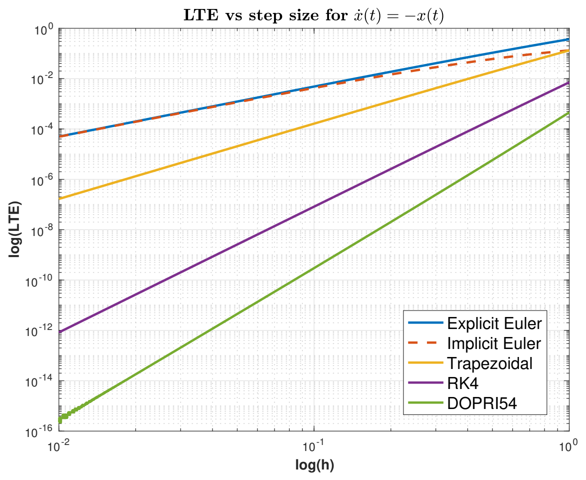
Figure 4: Global error over time for IVP2: $\ddot{x}(t) = x(t) \quad x(0) = 1 \quad \dot{x}(t) = 0$

1.4 Plot the local error at $t = t_0 + h$ as function of step size h.

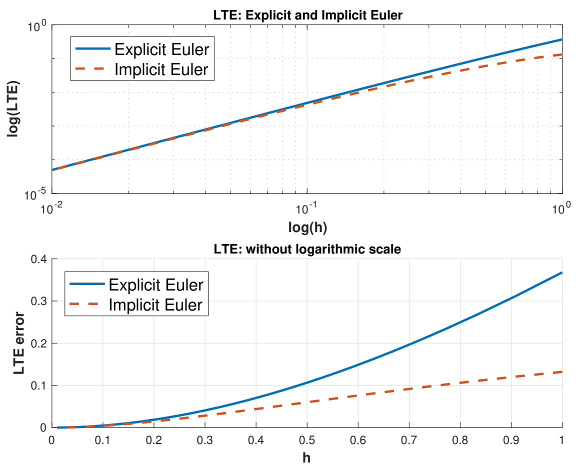
By using the Taylor decomposition, it is possible to bound the local error to

$$LTE = ch^p \tag{7}$$

where c is a constant (higgest values of the $p-th$ derivative in the domain, being q the order of the method, with $p = q + 1$. Hence, if the error is represented in a logarithmic scale for several step sizes, the slope of the line would be the same as the order of the solver.
In the figures 5 and 6 this idea can be appreciated. And the slope measurements collected in table 1 confirm that all methods behave as expected.

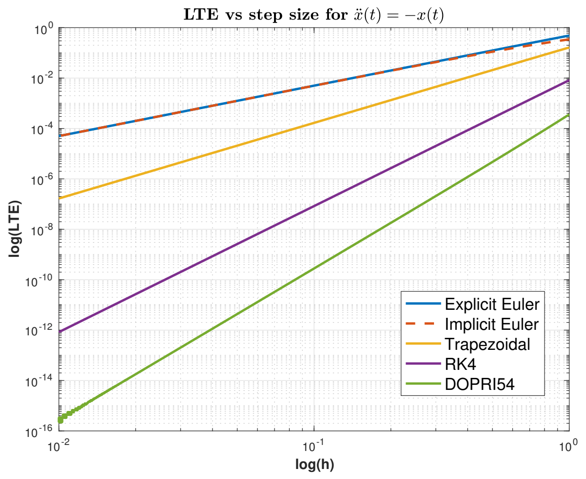


(a) All solvers

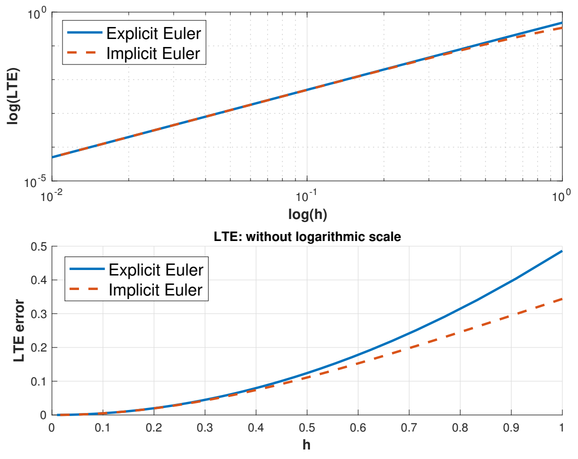


(b) Euler methods

Figure 5: LTE vs Step size for IVP1: $\dot{x}(t) = -x(t) \quad x(0) = 1$



(a) All solvers



(b) Euler methods

Figure 6: LTE vs Step size for IVP2: $\ddot{x}(t) = -x(t) \quad x(0) = 1 \quad \dot{x}(t) = 0$

Table 1: Error slope

Method	LTE order	Estimated slope	Theoretical slope
Explicit Euler	$O(h^2)$	1.9997	2
Implicit Euler	$O(h^2)$	1.9951	2
Trapezoidal	$O(h^3)$	2.9998	3
Classical Runge Kutta	$O(h^5)$	4.9999	5
DOPRI54	$O(h^6)$	6.0032	6

1.5 Plot the estimated local error by the method.

DOPRI54 is the only method that has embedded error estimation. Hence, step doubling for error estimation has been implemented for the rest. Let us analyze this methodology for Forward Euler's solver.

$$x_{n+1}^{(0)} = x_n + hf(t_n, x_n) \tag{8}$$

if we denote $x(t+h)$ as the true value, then the LTE is,

$$\begin{aligned}\tau_{n+1}^{(0)} &= x(t+h) - x_{n+1}^{(0)} \\ &= ch^2\end{aligned} \tag{9}$$

By the Taylor theorem we know that the LTE is proportional to a unknown constant, c. If we now compute the same point but taking a mid step,

$$\begin{aligned} x_{n+\frac{1}{2}} &= x_n + \frac{h}{2} f(t_n, x_n) \\ x^{(1)}_{n+1} &= x_{n+\frac{1}{2}} + \frac{h}{2} f(t_{n+\frac{1}{2}}, x_{n+\frac{1}{2}}) \end{aligned} \tag{10}$$

Then in the worst case the error is going to be the sum of both errors.

$$\tau^{(0)}_{n+1} = c\left(\frac{h}{2}\right)^2 + c\left(\frac{h}{2}\right)^2 = \frac{1}{2} ch^2 = \frac{1}{2} x^{(0)}_{n+1} \tag{11}$$

Then if we consider $x(t+h)$ as the real solution,

$$\begin{aligned} x(t+h) &= x(t+h) \\ x^{(1)}_{n+1} + \tau^{(1)}_{n+1} &= x^{(0)}_{n+1} + \tau^{(0)}_{n+1} \\ x^{(1)}_{n+1} - x^{(0)}_{n+1} &= \tau^{(0)}_{n+1} - \tau^{(1)}_{n+1} \\ x^{(1)}_{n+1} - x^{(0)}_{n+1} &= \tau^{(0)}_{n+1} - \frac{1}{2}\tau^{(1)}_{n+1} \\ x^{(1)}_{n+1} - x^{(0)}_{n+1} &= \frac{1}{2}\tau^{(1)}_{n+1} \end{aligned} \tag{12}$$

If we proceed in the same manner for the rest of solvers we arrive to

$$\begin{aligned} Trapezoid: \quad x^{(1)}_{n+1} - x^{(0)}_{n+1} &= \frac{3}{4}\tau^{(1)}_{n+1} \\ RK4: \quad x^{(1)}_{n+1} - x^{(0)}_{n+1} &= \frac{7}{8}\tau^{(1)}_{n+1} \\ DOPRI54: \quad x^{(1)}_{n+1} - x^{(0)}_{n+1} &= \frac{15}{16}\tau^{(1)}_{n+1} \end{aligned} \tag{13}$$

In the figures 7 and 8 it can be appreciated how the estimated error follows the real one. DOPRI54 error estimation is analyzed in the next section.

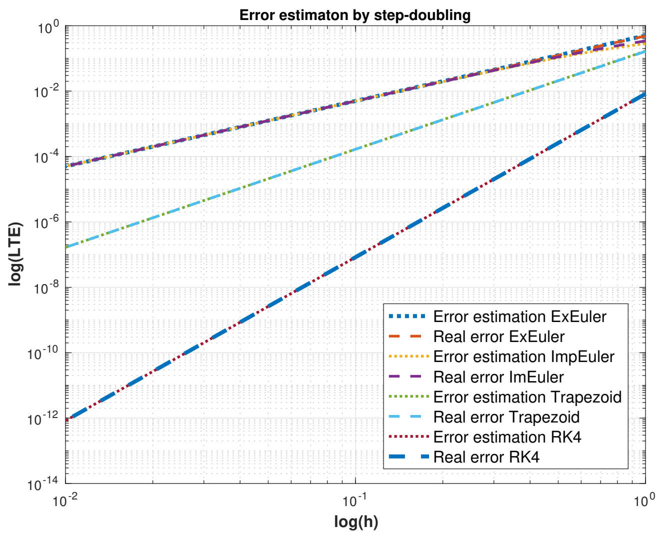


Figure 7: Real and estimated LTE vs step size for all solvers

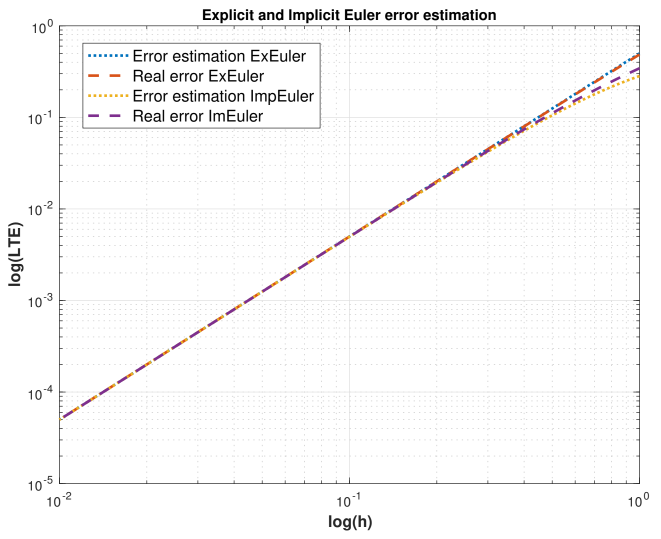


Figure 8: Real and estimated LTE vs step size for Euler solvers

1.6 Discuss the results and the quality of the error estimator for DOPRI54

As we can appreciate in the figure 9, the error estimation by step doubling is more precise than the embedded estimation, since it is the ssame order as our method, not like the embedded method, which has one order less. The price we are paying is computational, we need to compute 50% times the system function since we are computing double of points.

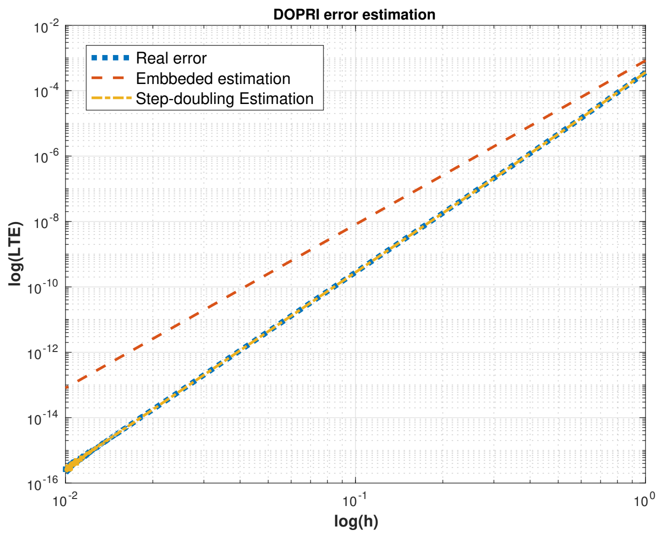


Figure 9: Embedded and Step-Doubling error estimation for DOPRI45

1.7 Can you come up with other (simple) methods for estimation of the local error

As explained in two previous sections, a simpler idea for error estimation is Step-doubling. Or maybe, since we are using several solver at a time anyway, we could substract them among them and get estimated of the error for free (given that need to compute all the solvers).

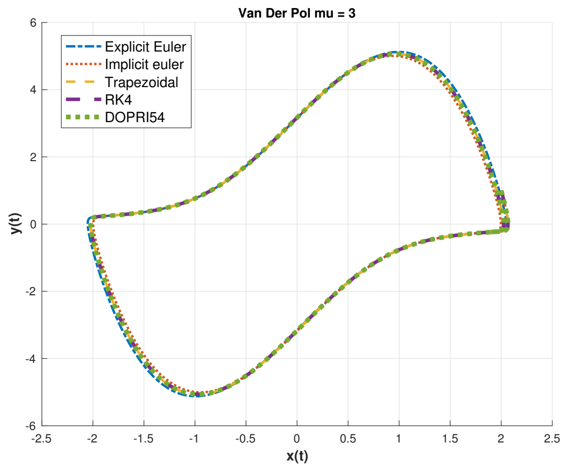
2 The Van der Pol System

2.1 Test your methods (1-5 from Problem 1) on the Van der Pol problem ($\mu = 3$). Plot the estimates of the errors as well as the solution.

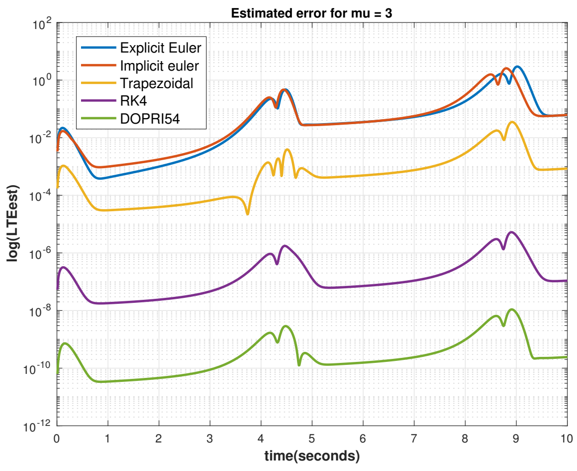
The Van der Pol problem can be written in multidimensional form:

$$\begin{aligned} \dot{x} &= y \\ \dot{y} &= \mu(1 - x^2)y - x \end{aligned} \tag{14}$$

In the figures 10, 11 and 12, it is shown the solution and error estimation for different step sizes.
The error present some peaks due the sudden change in the function, but all the methods are able to follow it.

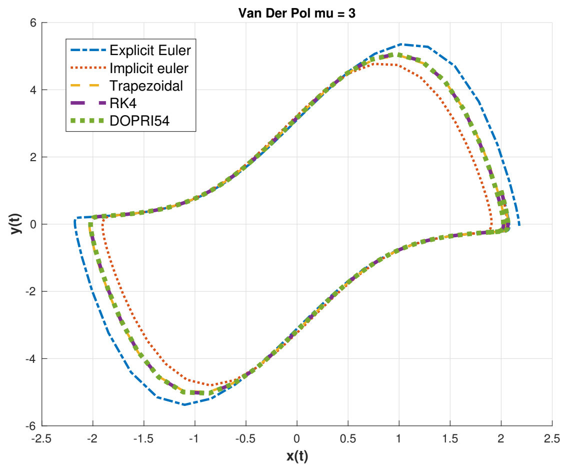


(a) Solution

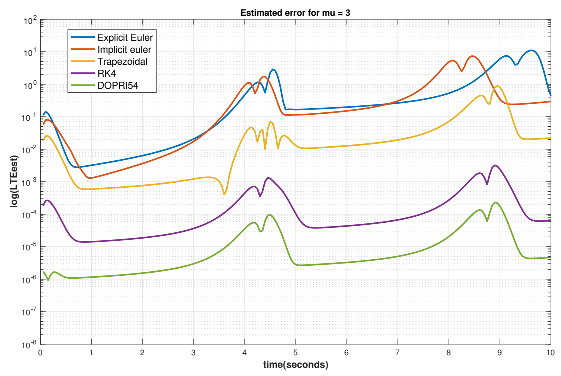


(b) Estimated LTE error

Figure 10: Van der Pol problem for $\mu = 3$ and $h = 0.1$

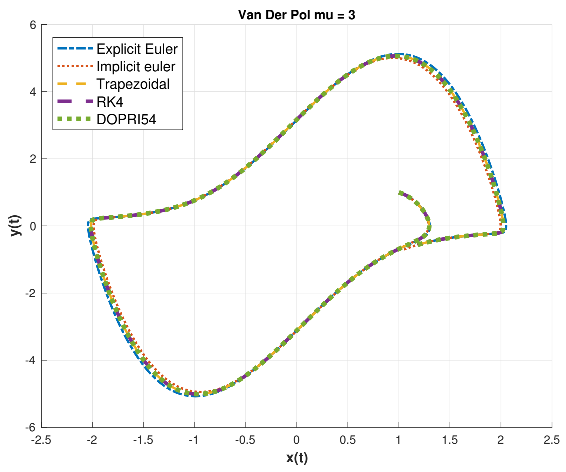


(a) Solution

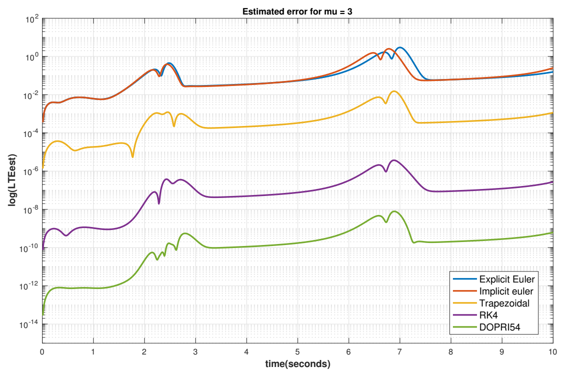


(b) Estimated LTE error

Figure 11: Van der Pol problem for $\mu = 3$ and $h = 0.05$



(a) Solution

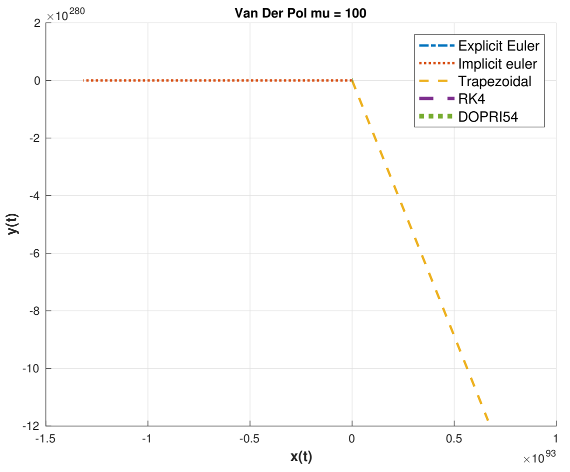


(b) Estimated LTE error

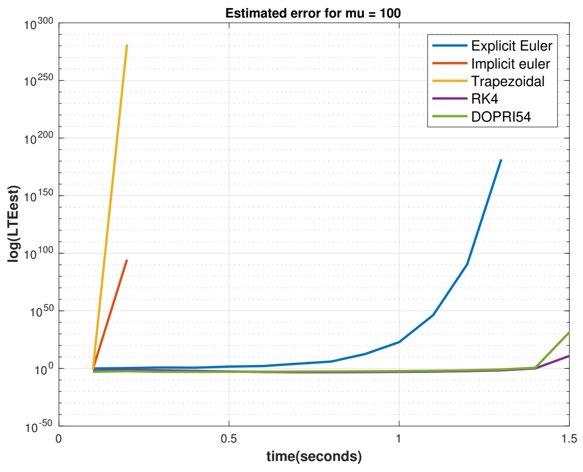
Figure 12: Van der Pol problem for $\mu = 3$ and $h = 0.01$

2.2 Test your methods on the Van der Pol problem with $\mu = 100$. Describe what happens.

When μ grows, the Van der Pol solution becomes more stiff. Around the values -2 and 2 in the x coordinate, there is a very abrupt change which make the solvers become unstable as the figure 13 represents, where a step size of $h = 0.1$ has been used.
Only when the step size is reduced to $h = 0.001$ the solvers are able to follow such change in the curve. Now in figure 14b it is even more obvious that the error increases when this sudden change occurs.

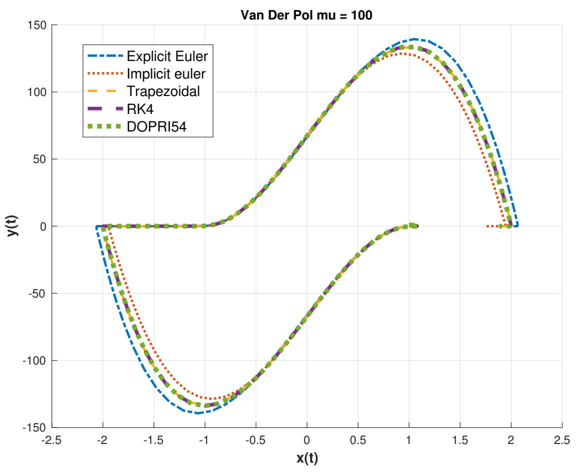


(a) Solution

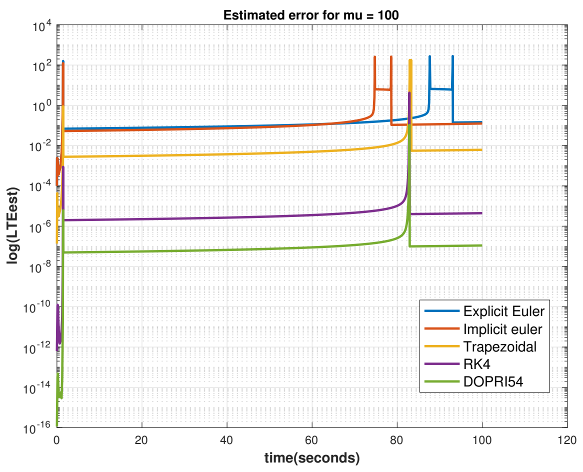


(b) Estimated LTE error

Figure 13: Van der Pol problem for $\mu = 100$ and $h = 0.1$



(a) Solution



(b) Estimated LTE error

Figure 14: Van der Pol problem for $\mu = 100$ and $h = 0.001$

3 Design your own Explicit Runge-Kutta Method

Design your own explicit Runge-Kutta method and apply it to the test equation and the Van der Pol equation.

3.1 Write up the order conditions for an embedded Runge-Kutta method with 3 stages. The solution you advance must have order 3 and the embedded method used for error estimation must have order 2.

The family of Runge-Kutta methods, is a family of One Step methods that allows us to construct ODE solvers, in which the estimation for the next step y_{t+1} is computed as a linear combination of a set of s-stage derivatives F_j given by the set of combination constants b_j, with $j = 1, ..., s$.

In general, an s-stage Runge-Kutta method for the ODE system $y' = f(t, y)$ being $y = y(t)$ an $n-th$ dimensional function of time, has the form:

$$y_{t+1} = y_t + h \sum_{j=1}^{s} b_j \mathbf{F}_j \tag{15}$$

Where the stage derivatives F_j are obtained from the evaluation of the function system in intermediate points $f(t_i, y_i)$ with $t_i = t + c_i$ and $y_i = \mathbf{Y}_i$.

$$\mathbf{F}_j = f(t + c_j h, Y_j) \tag{16}$$

The $s-$stage values Y_i are obtained from the system of equations as:

$$\mathbf{Y}_i = y_t + h \sum_{j=1}^{s} a_{ij} F_j = y_t + h \sum_{j=1}^{s} a_{ij} f(t + c_j h, Y_j) \tag{17}$$

The most general version of the Runge-Kutta methods, it is a family of implicit Runge-Kutta methods that requires us to solve a nxs implicit coupled nonlinear system of equations in each integration step. In carrying out a step we compute the s stage values Y_i with $i = 1, ..., s$.

Basically, in the Runge-Kutta method, we estimate the next value of the function, y_{t+1} as the previous value, y_t plus the step-size h multiplied by a linear combination of slopes F_j. The coefficients of this linear combination are the **b** coefficients. The slopes $\mathbf{F}_j$ are obtained evaluating the $f(t, y)$ function in several intermediate points $(t_i, y_i) -> (t + c_i h, \mathbf{Y}_i)$ where the coefficients **c** indicate the value at time t_i where we evaluate the derivative and the stage values Y_i are obtained from a linear combination of the slopes F_j given by the $\mathbf{a}_{ij}$ coefficients.

So the Runge-Kutta family has 3 sets of coefficients. The vector b, the matrix A and the vector c. We can represent the parameters of a given s-stage Runge-Kutta method by its Butcher tableau.

$$\begin{array}{c|cccc}
c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\
c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\
\vdots & \vdots & \vdots & & \vdots \\
c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}$$

The explicit version of the Runge-Kutta family imposes limitations on the parameter values, it makes the properties of the system less powerful but it decreases a lot the computational complexity since it does not contain implicit equations and many parameters are set to 0. For the method to be explicit, the value of a stage Y_i cannot depend on itself or any other following step. That means, that the a_{ij} coefficients for a explicit method must 0 for $j \geq i$. The simplified tableau is:

$$\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
\vdots & \vdots & \vdots & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}$$

In this exercise, we are asked for **3-stage** method, which means $s = 3$. The coefficients have to be chosen in such a way that the method has order 3, which means that our approximation error is bounded by $O(h^4)$

which at its core means that, the Taylor expansion of our estimation must have the same coefficients for the first, second and third derivative elements of the Taylor expansion of the original function $y(t)$.

What we can do is obtain the Taylor expansion of the method and see the conditions that the coefficient must follow in order for the coefficients of its derivatives to the same as for the expansion. For this purpose we can use the Chain Rule to express the successive derivatives in terms of the previous ones

$$\begin{aligned} y'(t) &= f(y(t)) \\ y''(t) &= f'(y(t))y'(t) \\ &= f'(y(t))f(y(t)) \\ y'''(t) &= f''(y(t))(f(y(t)), y'(t)) + f'(y(t))f'(y(t))y'(t) \\ &= f''(y(t))(f(y(t)), f(y(t))) + f'(y(t))f'(y(t))f(y(t)) \end{aligned} \tag{18}$$

Changing the nomenclature from $y'(t)$ to f, the Taylor expansion of the exact solution can be expressed as follows.

$$y_{t+1} = y(t_n) + hf + \frac{h^2}{2!}f' + \frac{h^3}{3!}f'' + O(h^4) \tag{19}$$

For the method to have order 1, we have condition that the coefficient of f our method must be 1. This leads to the condition:

$$\sum_{i=1}^{s} b_i = 1 \tag{20}$$

For the method to have order 2, we have condition that the coefficient of f' our method must be $\frac{1}{2}$. This leads to the condition:

$$\sum_{i=1}^{s} b_i c_i = \frac{1}{2} \tag{21}$$

For the method to have order 3, we have condition that the coefficient of f'' our method must be $\frac{1}{6}$. This leads to the conditions:

$$\sum_{i=1}^{s} b_i c_i^2 = \frac{1}{3} \tag{22}$$

$$\sum_{i=1}^{s} \sum_{k=1}^{s} b_i a_{ik} c_i = \frac{1}{6} \tag{23}$$

For the case of a 3 stage explicit Runge-Kutta, we have only 8 parameters $(a_{21}, a_{31}, a_{32}); (b_1, b_2, b_3); (c_2, c_3)$ since the the A matrix is a lower triangle. And we have the set of equation for c_j to fulfill the consistency conditions so:

$$c_i = \sum_{j=1}^{s} a_{ij} \tag{24}$$

This equation can be applied to the states 2 and 3, it is clear that $c_0 = 0$ since the first row of the A matrix is a zero row. So we have 8 parameters and 6 equations, 1 from $O(h)$, another from $O(h^2)$, 2 from $O(h^3)$ and 2 from the consistency condition values of c_i. We have 2 more variables than equations so there exists a two parameter family of methods of order three with three stages. We can select two coefficients and make the other six be determined by the conditions.

The following shows some examples of Bureau tables that fulfill these conditions.

$$\begin{array}{c|ccc} 0 & & & \\ \frac{1}{2} & \frac{1}{2} & & \\ 1 & -1 & 2 & \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array} \qquad \begin{array}{c|ccc} 0 & & & \\ \frac{2}{3} & \frac{2}{3} & & \\ \frac{2}{3} & 0 & \frac{2}{3} & \\ \hline & \frac{1}{4} & \frac{3}{8} & \frac{3}{8} \end{array} \qquad \begin{array}{c|ccc} 0 & & & \\ \frac{2}{3} & \frac{2}{3} & & \\ 0 & -1 & 1 & \\ \hline & 0 & \frac{3}{4} & \frac{1}{4} \end{array}$$

There are other conditions that we can apply to the choosing of these 2 free parameters, for example choosing the ones that minimize the coefficient of the error for Local Truncation error, calculated as:

$$TE = b_3 c_2 a_{3,2} + (b_2 c_2^2 + b_3 c_3^2) - \frac{1}{2} \tag{25}$$

The coefficients that minimize this error given the other constrains are $c_3 = 1$ and $c_2 = \frac{1}{2}$. Then we can apply the set of equations to obtain the values of the other parameters.
The resulting set of equations is:

$$\begin{aligned}
&b_1 + b_2 + b_3 = 1 \\
&b_2c_2 + b_3c_3 = \frac{1}{2} \\
&b_2c_2^2 + b_3c_3^2 = \frac{1}{3} \\
&b_3a_{32}c_2 = \frac{1}{6} \\
&c_2 = a_{23} \\
&c_3 = a_{31} + a_{32}
\end{aligned} \tag{26}$$

3.2 Derive the coefficients for the error estimator.

We can obtain an error estimation for the 3-stage, order 3 Runge-Kutta method in the following way. Using the same set of parameters as the original Runge-Kutta we have designed, we can change the b parameters to create another method, if we change these parameters and they fullfill the equations for order 1 and 2.

$$\sum_{i=1}^{s} b_i = 1$$

$$\sum_{i=1}^{s} b_i c_i = \frac{1}{2}$$

Then we have another Runge-Kutta method of order 2, and we did not need to compute any extra $f(t, y)$ so the computational complexity has not grown.

The substraction of both estimations, the original and the new one, will give an estimate of the error of the order 2 method.

So in our case, the new values of $\hat{b}$ must fulfil the equations

$$\begin{aligned}
\hat{b}_1 + \hat{b}_2 + \hat{b}_3 &= 1 \\
\hat{b}_2c_2 + \hat{b}_3c_3 &= \frac{1}{2}
\end{aligned} \tag{27}$$

We select for example $\hat{b}_2 = \frac{1}{2}$ and so, using the equations we have that $\hat{b}_1 = \frac{1}{4}$ and $\hat{b}_3 = \frac{1}{4}$. So the local error estimation will be:

$$LTE = \hat{y}_{n+1} - y_{n+1} = h\sum_{j=1}^{s}(\hat{b}_i - b_i)F_i = h\sum_{j=1}^{s} d_iF_i \tag{28}$$

Where d_i are the coefficients to compute the error, $d_i = \hat{b}_i - b_i$.
In our case, we have $d_1 = \frac{1}{12}, d_2 = -\frac{1}{6}, d_3 = \frac{1}{12}$

3.3 Write up the Butcher tableau for your method and the corresponding equations defining the method.

From the derivations of the above sections we have the Butcher tableau:

$$\begin{array}{c|ccc}
0 & 0 & 0 & 0 \\
\frac{1}{2} & \frac{1}{2} & 0 & 0 \\
1 & -1 & 2 & 0 \\
\hline
b & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \\
\hline
d & \frac{1}{12} & -\frac{1}{6} & \frac{1}{12}
\end{array}$$

The equations for the solver are the following, the intermidiate stages are:

$$y_{t+1} = y_t + h\sum_{j=1}^{s} b_jF_j$$

Where the stage derivatives F_j are obtained from the system of equations as:

$$\begin{aligned}
\mathbf{Y}_1 &= y_t \\
\mathbf{F}_1 &= f(t + c_1 h, Y_1) \\
\\
\mathbf{Y}_2 &= y_t + h(a_{21} F_1) \\
\mathbf{F}_2 &= f(t + c_2 h, Y_2) \\
\\
\mathbf{Y}_3 &= y_t + h(a_{31} F_1 + a_{32} F_2) \\
\mathbf{F}_3 &= f(t + c_3 h, Y_3)
\end{aligned} \tag{29}$$

The function in the next instant is estimated as:

$$y_{t+1} = y_t + h(b_1 F_1 + b_2 F_2 + b_3 F_3)$$

And the estimation of the error for the order 2 method is:

$$LTE = y_t + h(d_1 F_1 + d_2 F_2 + d_3 F_3)$$

The code to implement this methods is a modification of the code given by John Bagterp.

3.4 Test your solver on the test equation with $\lambda = -1$.

The following figure shows 4 graphs related to the testing of the method with the test equation. The step-size has been chosen to be $h = 0.01$ so that the system is stable and somewhat accurate.

- The first graph shows the actual function $y(t)$ and our estimated function y_t obtained from the solver. As we can appreciate, they are too close to each other to differentiate them in the graph. So far the solver seems to work in this easy example.
- The second graph shows some aspects of the **local truncation error** of the system in several ways. First, the 2 decreasing lines show the actual local truncation error LE_n and the estimated truncation error $\hat{LE}_n$ in a logarithmic scale. As we can see, they both decrease exponentially and the estimated local error is bigger than the real one, since the method used for estimation has one degree less than the actual method. The equation for the real truncation error is:

$$\begin{aligned}
LE_{n+1} =& y_n[e^z - R(z)] \\
=& y_n E(z)
\end{aligned} \tag{30}$$

  Being $z = h\lambda$ just the product of the step-size and the exponent of the function. $R(z)$, as we will see later, is the characteristic polynomial of the Runge-Kutta method. It is the coefficient between the next estimated value of the function, and the previous one. In this case:

$$R(z) = \frac{y_{n+1}}{y_n} = R(\lambda h) = (1 + \lambda h + \frac{1}{2}\lambda^2 h^2 + \frac{1}{6}\lambda^3 h^3) \tag{31}$$

  In the graph, we can also see the coefficient of the truncation error $E(z)$, this is, at every step, the difference between our estimation and the real value of the function, divided by the actual value of the function.

$$E(z) = [e^z - R(z)] \tag{32}$$

  We can see how, of course, the coefficient is the same for all instants. It is the difference between the real geometric update $e^{h\lambda}$ and our estimate $R(h\lambda)$

- The third graph shows the global error, the difference between $y(t)$ and y_t. This can be analytically found to have the form:

$$\begin{aligned} Error =& y_n - y(t_n) \\ =& y_0[R(z)^n - e^{(t_n - t_0)}] \\ =& y_0[R(z)^n - e^{(nz)}] \end{aligned} \tag{33}$$

As we can see, if $|R(z)| > 1$, then our error will grow exponentially and the system will not be stable.

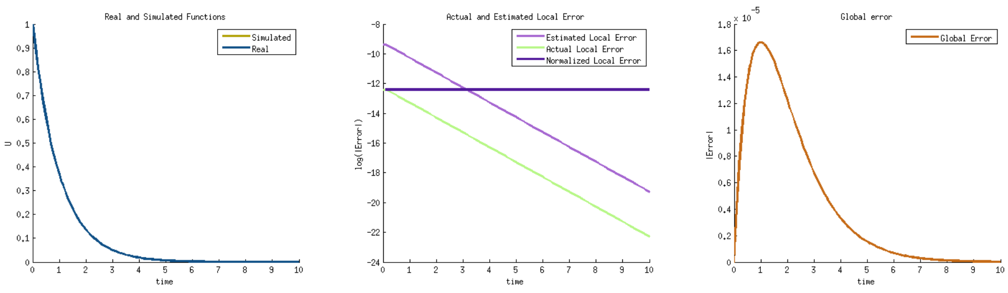


Figure 15: Testing of the RK3 with the test equation

3.5 Verify the order of your method by plotting the local error as function of step size (for the test equation).

To verify the order of the method we have to check what is the relationship between the actual error e_L and estimated error $\hat{e}_L$ with the step size h. This relationship will be of the form:

$$e_L \approx Ch^{q+1}$$

Being C some constant and q the order of the system. A possible way to visualize and compute q is to first take logarithm of e_L, so that we have:

$$log(e_L) = log(Ch^{q+1}) = log(C) + (q+1)log(h)$$

Which is a linear function of $log(h)$, so the slope of this function will be $q+1$, which indicates the order of the method.

In the next graph we have plotted the logarithm of the actual and estimated local errors for different step sizes, against the logarithm of the step size. As we can see, in the graph, the slope of the actual local error is 4, which means the method has order 3, and the slope of the estimated local error is 3, so it has order 2.

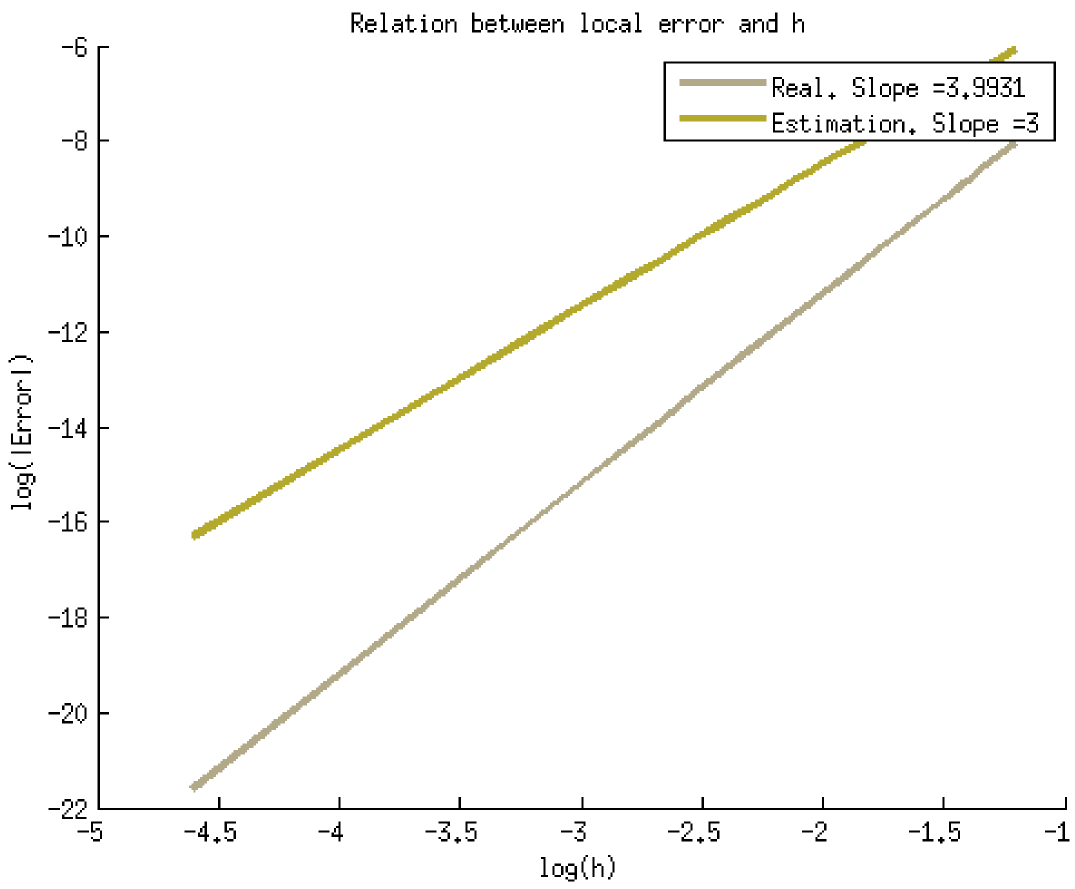


Figure 16: Relation between local error and stepsize for the RK3 method

3.6 Compute $R(h\lambda)$ for your method and make a stability plot of your method.

The $R(h\lambda)$ polynomial is obtained as the ratio between the estimation of the function at time t_n and the estimation at t_{n+1} when the right hand side function is given as $f(y,t) = \lambda y$.

$$R(z) = R(h\lambda) = \frac{y_{n+1}}{y_n}$$

For our Runge-Kutta method we have:

$$\begin{aligned} k_1 =& \lambda y_n \\ k_2 =& \lambda(y_n + a_{2,1}hk_1) = \lambda(1 + c_2 h\lambda)y_n \\ k_3 =& \lambda(y_n + a_{3,1}hk_1 + a_{3,2}hk_2) \\ =& \lambda(1 + a_{3,1}h\lambda + a_{3,2}h\lambda(1 + c_2 h\lambda)y_n \\ =& \lambda(1 + c_3 h\lambda + a_{3,2}c_2 h_2\lambda^2)y_n \\ y_{n+1} =& y_n + h(b_1k_1 + b_2k_2 + b_3k_3) \\ =& y_n(1 + h\lambda(b_1 + b_2 + b_3) + h^2\lambda^2(b_2c_2 + b_3c_3) + h^3\lambda^3 b_3 a_{3,2}c_2) \end{aligned} \tag{34}$$

Applying the conditions for order 3 we obtain the result

$$\begin{aligned} y_{n+1} =& y_n(1 + \lambda h + \frac{1}{2}\lambda^2h^2 + \frac{1}{6}\lambda^3h^3) \\ =& y_n P_3(h\lambda) \end{aligned} \tag{35}$$

In a more general fashion, a Runge-Kutta method of order q will have the form:

$$\begin{aligned} y_{n+1} =& y_n + zy_n\mathbf{b}^T(\mathbf{I} - z\mathbf{A})^{-1}\mathbf{I} \\ =& y_n[1 + z\mathbf{b}^T(\mathbf{I} + z\mathbf{A} + z^2\mathbf{A}^2 + z^3\mathbf{A}^3 + ... + z^q\mathbf{A}^q)\mathbf{I} \end{aligned} \tag{36}$$

Using the order conditions, the equation simplifies to:

$$\begin{aligned} y_{n+1} =& y_n[1 + \frac{z^2}{2!} + \frac{z^3}{3!} + ... + \frac{z^q}{q!}) \\ y_{n+1} =& y_n R(z) \end{aligned} \tag{37}$$

We can see the previous equation as an autoregressive Infinity Impulse Response (IIR) system, this system will be stable if the coefficient $|R(z)| < 1$, otherwise, the value of y_n will grow exponentially (geometrically). The following graph contains the stability graph for our Runge-Kutta method, where $z = h\lambda$ belongs to the complex domain. In the plot, the parts where $|R(z)| > 1$ are saturated to 1, but they obviously have bigger values.

- The first graph indicated the absolute value of the auto-regression constant $|R(z)|$. If its value is bigger or equal to 1, then the system will diverge, it will not be stable. As we can see, the system is not A-Stable because the left part of the response is not lower than 1. We can see how the stability region is bigger than the unit circle, due to the higher order components of the characteristic polynomial $R(z)$.
- The second graph indicates the module of the relative local error,$|E(z)|$. This is the difference between the real change e^z and the autoregression constant $|R(z)|$ of the method. As we can see, the bigger is z, the bigger the error, with some deviation to the right.

$$\begin{aligned} LE_{n+1} =& y_n[e^z - R(z)] \\ =& y_n E(z) \end{aligned} \tag{38}$$

- The third and forth graphs are absolute values of the estimation of the local error of the method $\hat{LE}_{n+1}$ and the difference between the actual truncation error and the estimated one, $LE_{n+1} - \hat{LE}_{n+1}$. As we can see the estimation of the error has the same shape as the actual error.

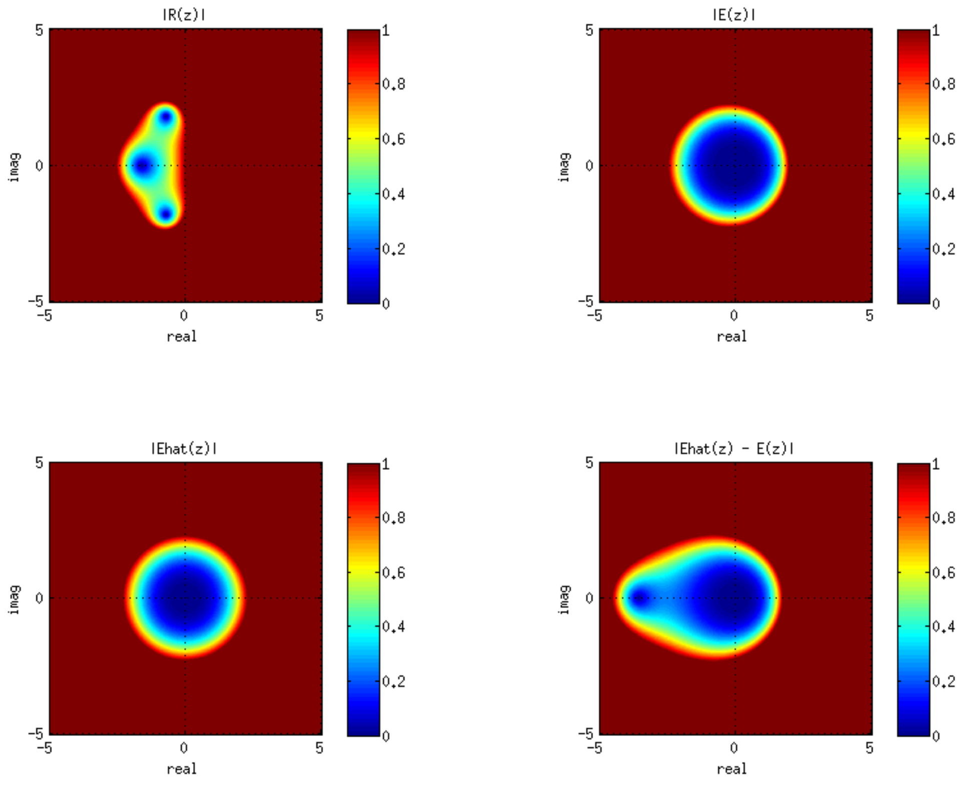


Figure 17: Stability plots of RK3

The code to compute these plots is a modification of the code given by John Bagterp.

3.7 Test your solver on the van der Pol equation with $\mu = 3$. Compare the solution you get with the solution you get when using ode15s.

The next figure shows the approximation of the ode15s solver and our implementation of the RK3 method implemented. The step size is $h = 10^2$. As we can see, the two methods output pretty much the same functions. We cannot easily differentiate between them.

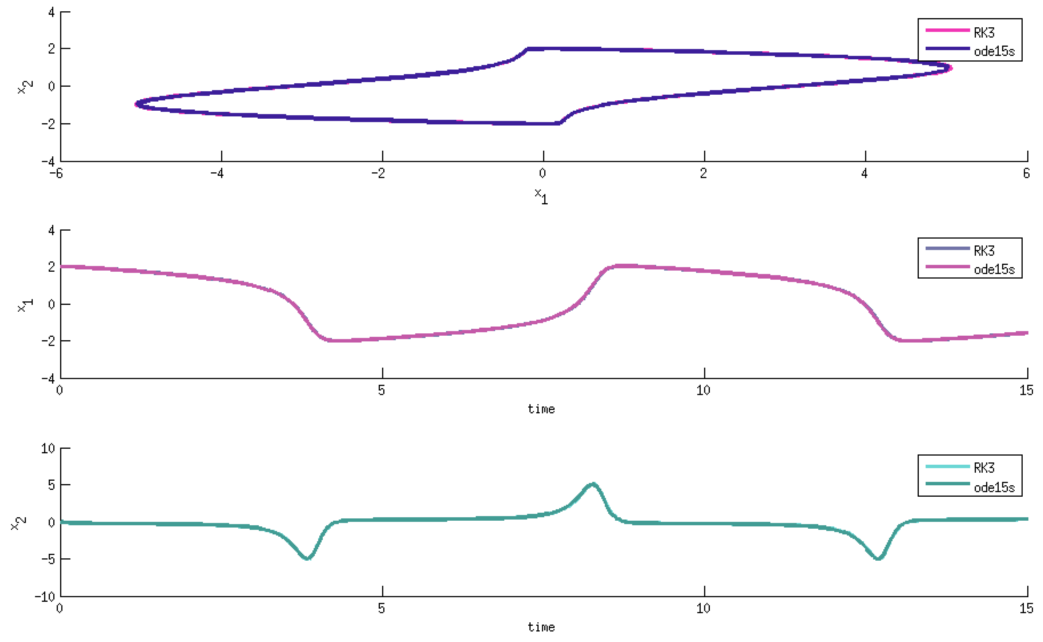


Figure 18: Solution of the solvers for the van der Pol equation with $\mu = 3$

The next graphs shows the estimation error of the RK3 method for the two individual dimensions and also for their module. As we can see, the estimation of the error is significantly bigger in those instants where the signals vary rapidly, which makes sense. We can almost conclude that our implementation is correct.

We also mention that we would have showed the subtraction between the estimate of the ode15s and the RK3 but since the stepsize of the ode15s is adaptative, the comparison required interpolation and we did not proceed.

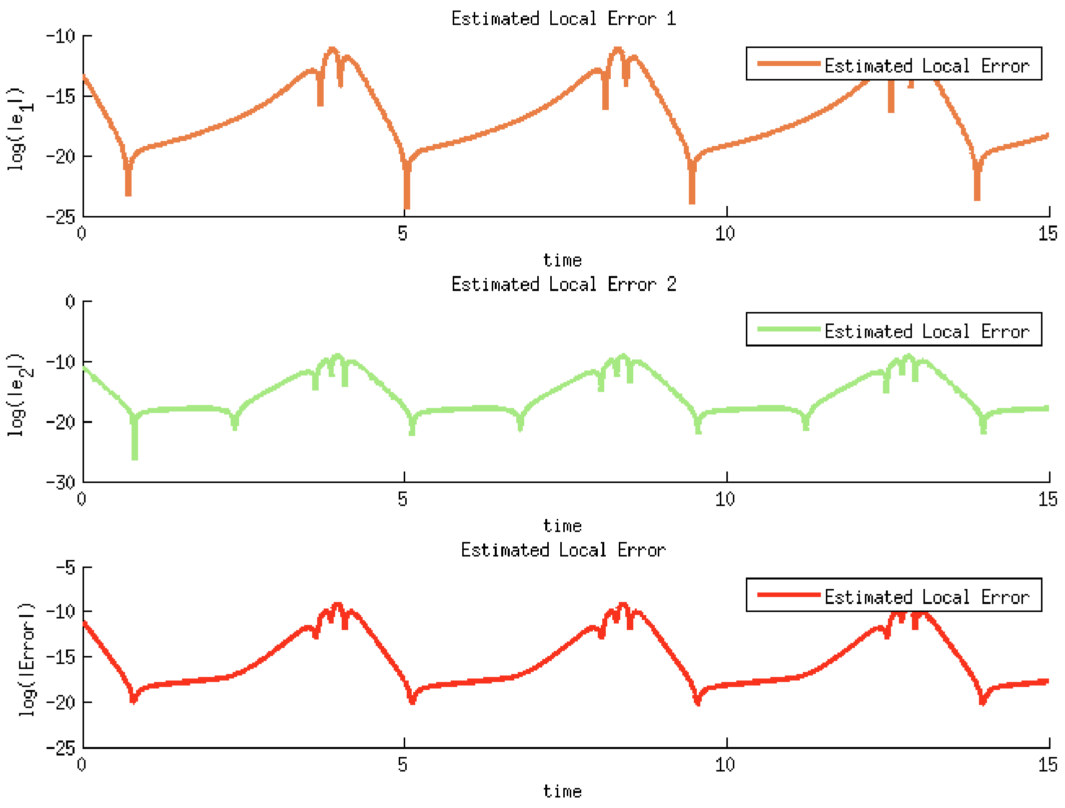


Figure 19: Estimation of the error of the RK3 solver for the van der Pol equation with $\mu = 3$

4 Step Size Controller

4.1 Implement an asymptotic step size controller

4.1.1 The P-controller

Again we know that our local error can be approximated as,

$$LTE \approx ch^p \tag{39}$$

now it is necessary to equal the LTE to a certain tolerance τ. This new τ will be associated to a certain $\hat{h}$.

$$\tau \approx c\hat{h}^p \tag{40}$$

and for changing our current h to meet the tolerance,

$$\frac{\tau}{LTE} \approx \frac{c\hat{h}^p}{ch^p} = \frac{\hat{h}}{ch^p} \tag{41}$$

then we should be able to satisfy the tolerance for the following step size,

$$\hat{h} = \sqrt[p]{\frac{\tau}{LTE}}h \tag{42}$$

However this controller is very aggressive and sometimes we might pass the tolerance. In order to solve this we introduce a reducing factor $\epsilon = 0.8$

$$\hat{h} = \epsilon\sqrt[p]{\frac{\tau}{LTE}}h \tag{43}$$

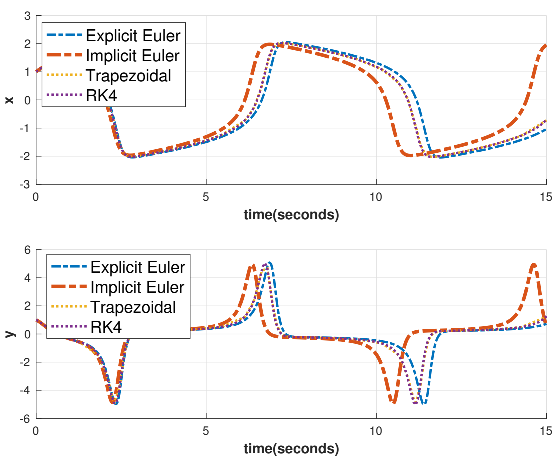
4.1.2 Defining the tolerance

Assuming that a error estimation is available, like step doubling or embedded method, we would like to impose to our system an absolute and relative tolerance over it at each step.

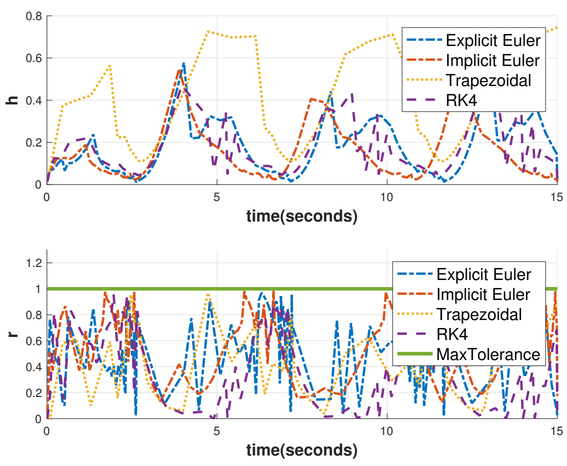
$$|e_i| \leq abstol \quad \frac{|e_i|}{|x_i|} \leq reltol \tag{44}$$

we can collet both terms in a single term using the supremum norm,

$$\tau = ||e_i||_\infty = \frac{|e_i|}{abstol + reltol|x_i|} \tag{45}$$

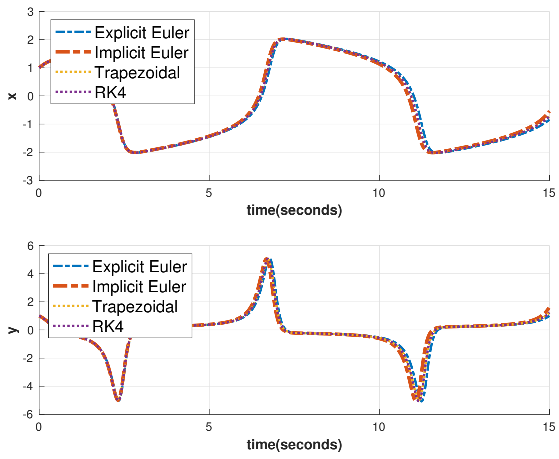


(a) Solution

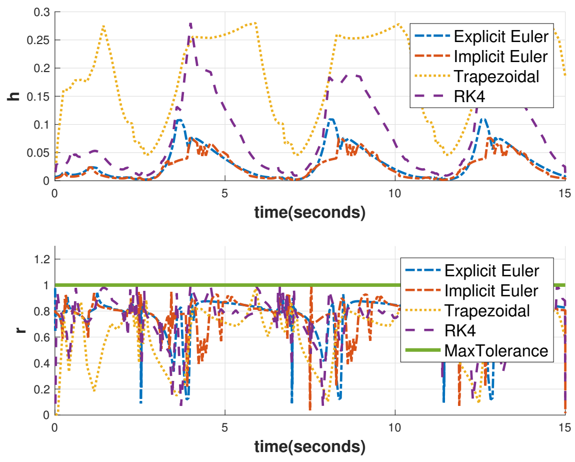


(b) Tolerance(r) and step size

Figure 20: Van der Pol problem for $\mu = 3$ with adpative step and $RelTol = AbsTol = 10^-3$

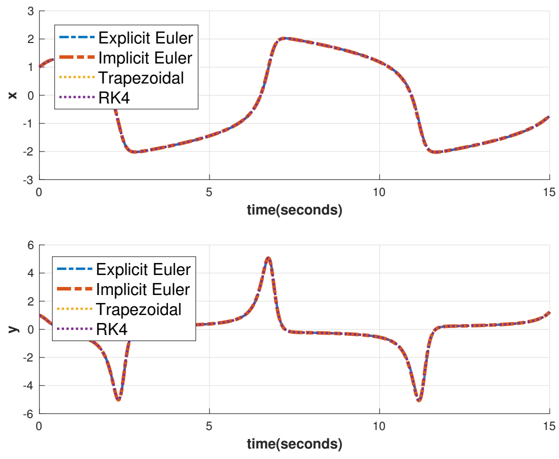


(a) Solution

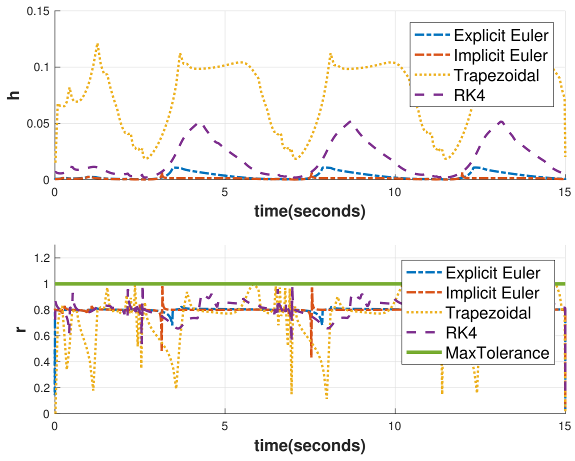


(b) Tolerance(r) and step size

Figure 21: Van der Pol problem for $\mu = 3$ with adpative step and $RelTol = AbsTol = 10^-5$



(a) Solution



(b) Tolerance(r) and step size

Figure 22: Van der Pol problem for $\mu = 3$ with adpative step and $RelTol = AbsTol = 10^-7$

In the figures 20,21 and22 we can see the solution for the adaptive step size solvers as well as the change of step size and estimated error. As expected, with a more restrictive tolerance the solution is more accurate. However, the number of steps and function calls grow, as the table 2 reflect.

It is important to notice from the figures 20b,21b,22b that the tolerance(green line) it is never surpassed.

Table 2: Number of function calls

Method	$Tol = 10^-3$	$Tol = 10^-5$	$Tol = 10^-7$
Explicit Euler	417	2852	28250
Implicit Euler	7112	19417	224302
Trapezoidal	9684	7863	22102
Classical Runge Kutta	739	1539	3523

4.2 Implement a PI step size controller

The PI controller developed by Gustaf Søderlind [2] is as follow,

$$\hat{h} = \epsilon h \left(\frac{\tau}{e_n}\right)^a * \left(\frac{\tau}{e_{n-1}}\right)^b * \left(\frac{h_n}{h_{n-1}}\right)^- c \tag{46}$$

where $a = 1/6$, $b = 1/6$ and $c = 1/2$

5 ESDIRK23

The family of ESDIRK methods are characterized by having a lower triangular A matrix with non-zero values in the diagonal except for the element a_{11}. Their parameters' table is as follows:

$$\begin{array}{c|cccc} c_1 & 0 & 0 & \cdots & 0 \\ c_2 & a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

This family is designed to have some of the nice stability properties of the implicit Runge-Kutta methods without having to solve in each integration step a system of nxs coupled nonlinear equations. For the ESDIRK methods, the internal stages decouples in a such a way that the iterations only need to solve an implicit equation (the first step is just an explicit equation).

So, in the ESDIRK methods, the equations are decoupled so we can calculate the stages Y_i one at a time. The first one is an explicit function, and for the rest we have implicit functions of just one variable, so we can solve them normally with the Newton's method.

From the derivations of the above sections we have the Butcher tableau:

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ c_2 & a_{22} & a_{22} & 0 \\ c_3 & a_{31} & a_{32} & a_{33} \\ \hline b & b_1 & b_2 & b_3 \\ \hline b & b_1 & b_2 & b_3 \end{array}$$

In the ESDIRK family, we put constraints as well in the auto-regressive coefficients a_{ii}, they are all equal to γ. We desire this family to have the L-stability of the implicit methods. In the following we will elaborate a little on the derivation of the parameters of the ESDIRK23 to fulfil the L-stability requirement.
The stability function of the 3-stage stiffly accurate ESDIRK integration scheme is

$$R(z) = \frac{P(z)}{Q(z)} = \frac{1 + (b_1 + b_2 - \gamma)z + (a_{21}b_2 - b_1\gamma)z^2}{(1 - \gamma z)^2} \tag{47}$$

To have L-stability, the numerator order must be less than the denominator order in the stability function. It would be enough that the numerator coefficient of the z^2 is 0.

$$a_{21}b_2 - b_1\gamma = 0$$

The consistency requirements for the ESDIRK23 scheme are:

$$\begin{aligned} c_2 &= a_{21} + \gamma \\ 1 &= b_1 + b_2 + \gamma \end{aligned} \tag{48}$$

The order conditions to have order 3 are:

$$\begin{aligned} Order1 :& b_1 + b_2 + \gamma = 1 \\ Order2 :& b_2c_2 + \gamma = 1 \\ Order3 :& b_2c_2^2 + \gamma = 13 \\ & 2b_2c_2\gamma + \gamma^2 = 16 \end{aligned} \tag{49}$$

As we can see, some of the order and consistency conditions are the same. We have a system of 5 nonlinear equations, and 5 unknown variables $c_2, a_{21}, \gamma, b_1, b_2$. The solutions of the system are two, corresponding to $\gamma = 3 \pm 6\sqrt{3}$. None of them are L-stable.

Knowing this imposition, instead of having an order 3 method, in the ESDIRK23 we construct a method of order 2. In addition we require that stage 2 has order 2. Gathering the The L-stability condition, the order 1 and 2 conditions, and the conditions for stage order 2 of stage 2, we have a system of 5 nonlinear equations with 5 unknown variables $c_2, a_{21}, \gamma, b_1, b_2$. This system has two solutions, corresponding to $\gamma = 2 \pm 2\sqrt{2}$.

From which we choose the solution Both of them are A-stable. However, only the solution corresponding to $\gamma = 2 \pm 2\sqrt{2}$ which has $0 < c_2 < 1$. With these choice, we have the method with parameters:

$$\begin{aligned} \lambda &= a_{21} = 2 - \sqrt{2}2 \\ c_2 &= 2\lambda = 2 - \sqrt{2} \\ b_1 &= b_2 = 1 - 2\lambda = -\sqrt{4}2 \end{aligned} \tag{50}$$

So we finally have a stiffly accurate, L-stable 3-stage ESDIRK structure of order 2. Now we design an embedded method to estimate the error. This method should have order 3 and it is determined by the order conditions. Given the previous selection of A, c parameters, the only possible choice is:

$$\begin{aligned} \hat{b}_1 =& \frac{6\gamma - 1}{12\gamma} \\ \hat{b}_2 =& \frac{1}{12\gamma(1 - 2\gamma)} \\ \hat{b}_3 =& \frac{1 - 3\gamma}{3(1 - 2\gamma)} \end{aligned} \tag{51}$$

which is neither A- nor L-stable. This ESDIRK method, called ESDIRK23, so the estimated error will not have an A stable region of convergence.

5.1 Implement ESDIRK23 with fixed step size.

In order to implement the ESDIRK23 method with fixed step we modified a more complete version of ESDIRK23 implementation from the file ESDIRK.m provided by John Bagterp. The modification concerns the removal of the sections dealing with step size control, rejecting or accepting the step, the policy for updating the Jacobian matrix and the fact that $g = x$ now. We also store the prediction error and put it as an output variable so that we can see an estimation of the error.

As discussed before, the main issue with the implementation is the resolution for the steps grater than 2, of an implicit equation, which is solved by Newton's method. Other than that, the logic will be the same as for the explicit Runge-Kutta methods except that, due to using the Newton's method for solving the implicit equations of the steps, we need to check weather or not the iterations converged and update the jacobian matrix.

5.2 Test your implementation on the Van der Pol equation with $\mu = 3$ and $\mu = 100$. Compare the solution and the number of function evaluations with your own Explicit Runge-Kutta method.

For the Van der Pol equation with $\mu = 3$ the next graphs show the solution and error estimates for both methods using a step size of $h = 0.001$ and time span from $t = 0$ to $t = 15$. As we can see, both systems are able to estimate the function correctly, their error estimation is similar, being lower for the ESDIRK23 system.

The number of function evaluations for the RK3 method is $Nf_{RK3} = 3 * Nsteps = 3 * 15001 = 45.003$ and the number for the ESDIRK method is $Nf_{ESDIRK23} = 46.048$. Both numbers are very similar, being the ESDIRK23 bigger due to more iterations in the Newton approximation of the implicit functions.

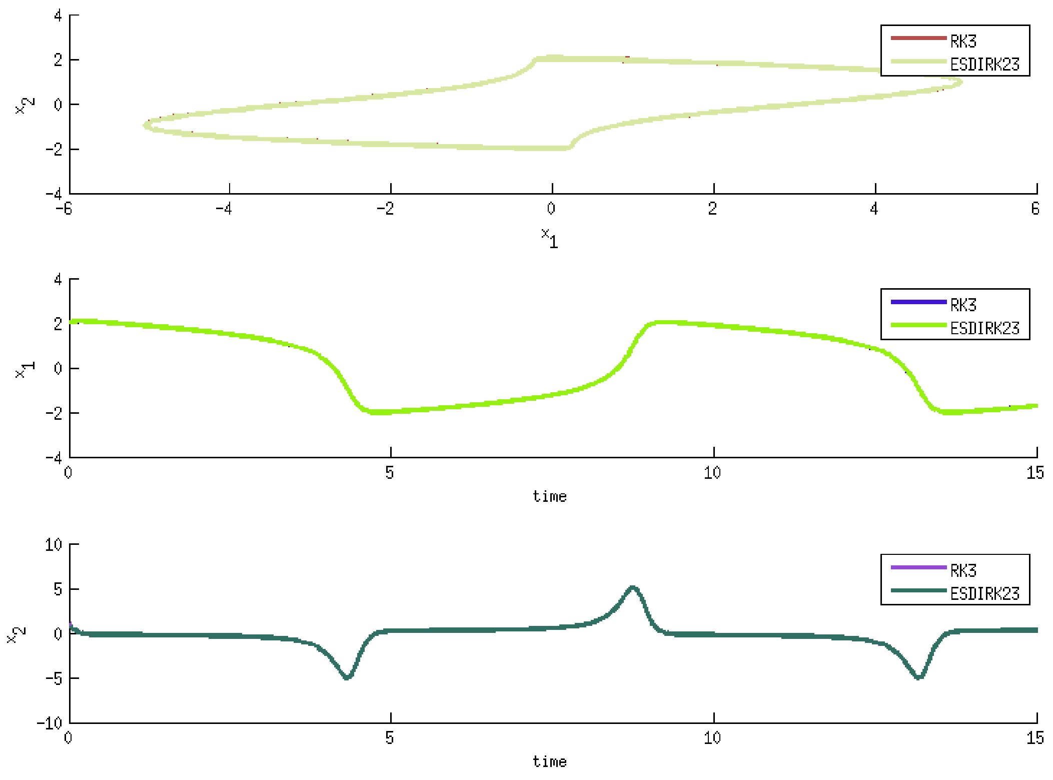


Figure 23: Solution of the solver for the methods RK3 and ESDIRK23 for the VanDerPol system with $\mu = 3$

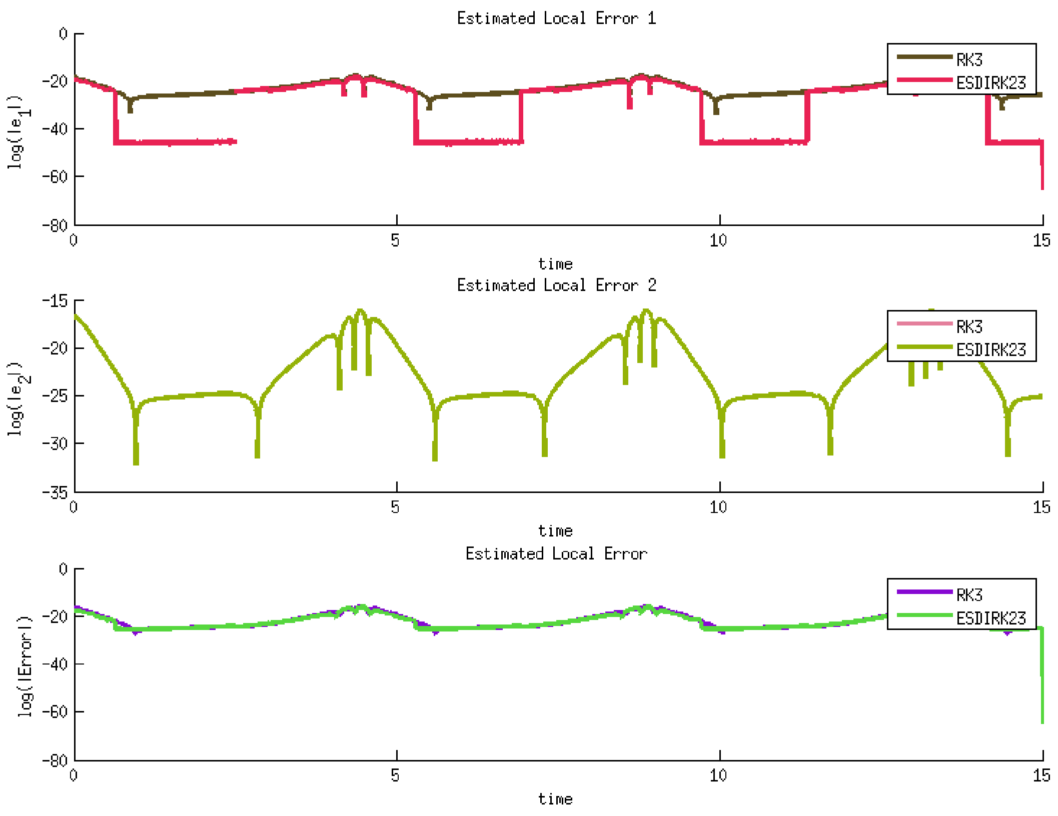


Figure 24: Estimation of the error for the methods RK3 and ESDIRK23 for the VanDerPol system with $\mu = 3$

For the Van der Pol equation with $\mu = 100$ the next graphs show the solution and error estimates for both methods using a step size of $h = 0.001$ and time span from $t = 0$ to $t = 300$. As we can see, both systems are able to estimate the function correctly, their error estimation is similar, being lower for the ESDIRK23 system.

The number of function evaluations for the RK3 method is $Nf_{RK3} = 3 * Nsteps = 3 * 300.001 = 900.003$ and the number for the ESDIRK method is $Nf_{ESDIRK23} = 605.552$. In this case the ESDIRK23 method requires only 2 thirds of the function evaluations of the RK3.

Here we can see how the the implicit methods are better for stiff problems. We can see how the error estimate is also smaller. This is due to the fact that due to implicit nature of the ESDIRK23, it can foresee big changes of the function, one time instant before the implicit methods, thus reducing the error significantly if the system varies quickly. It is appreciated that the estimation of the error is much smaller when the system is slowly changes and it peaks when the system has high volatility.

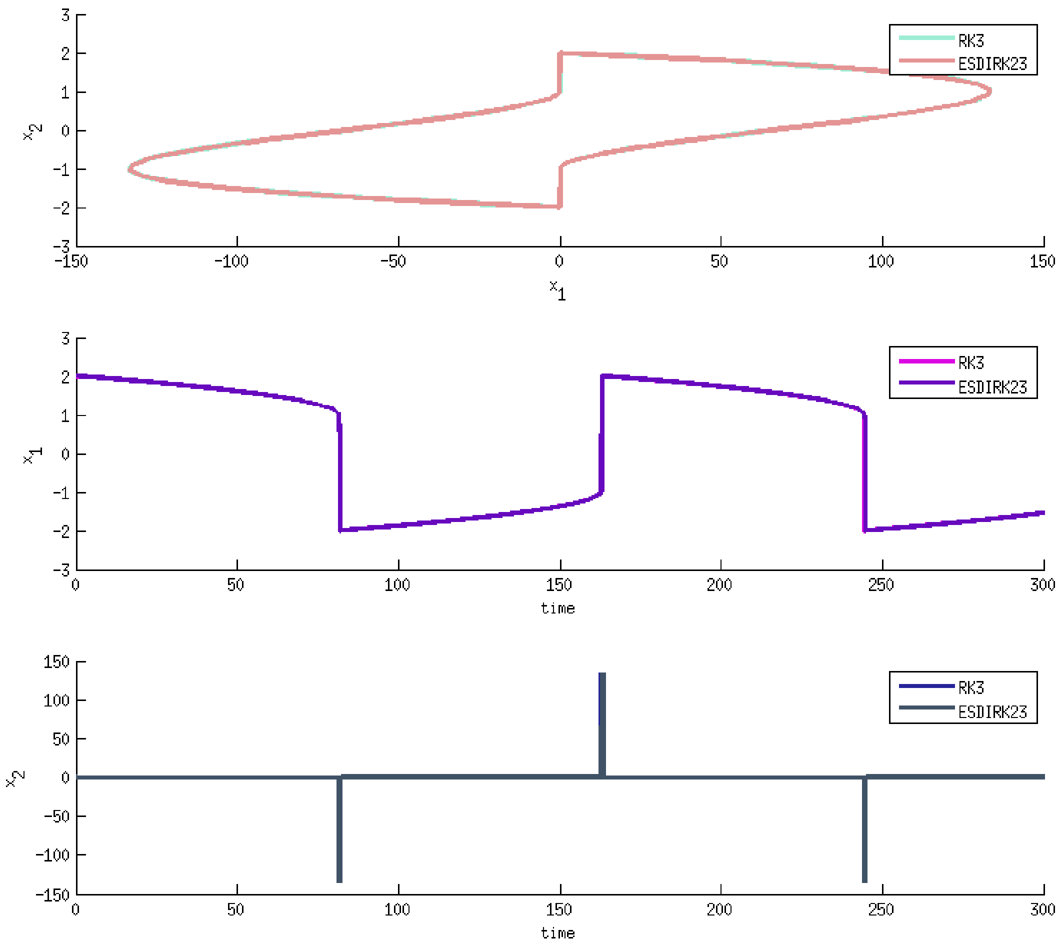


Figure 25: Solution of the solver for the methods RK3 and ESDIRK23 for the VanDerPol system with $\mu = 100$

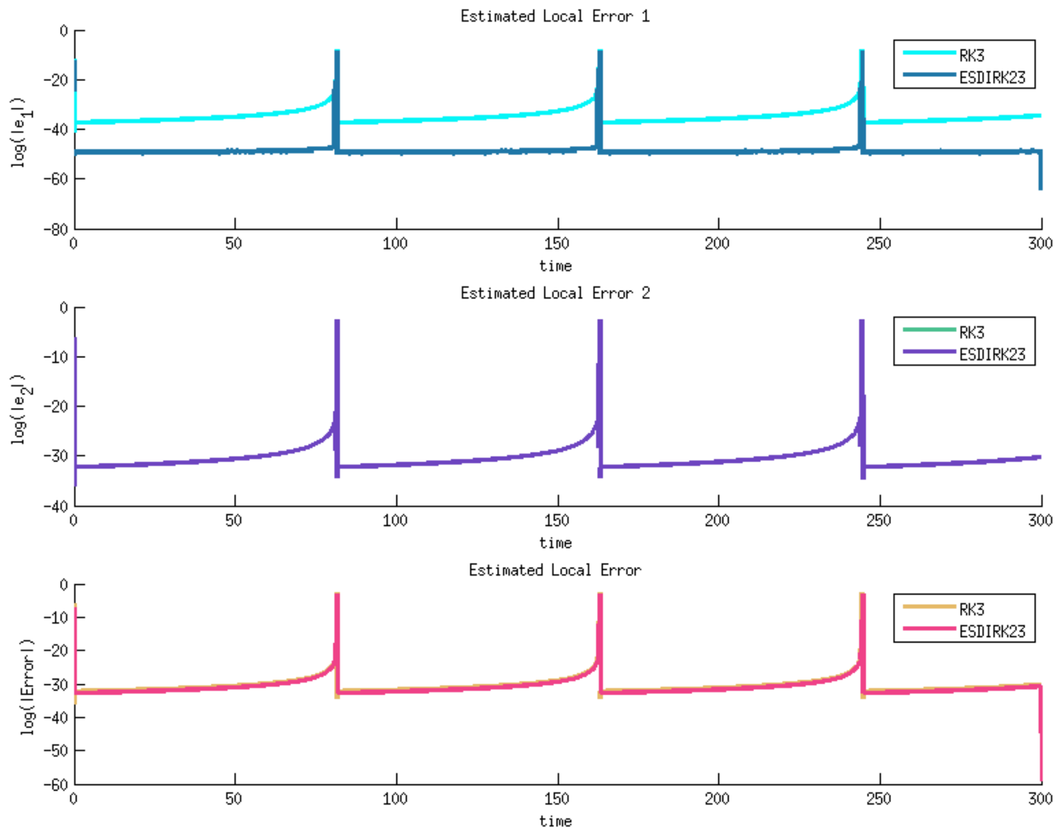


Figure 26: Estimation of the error for the methods RK3 and ESDIRK23 for the VanDerPol system with $\mu = 100$

5.3 Plot the stability region of the ESDIRK23 method. Is it A-stable? Is it L-stable? Discuss the practical implications of the stability region of ESDIRK23.

The following graph contains the stability graph for our Runge-Kutta method, where $z = h\lambda$ belongs to the complex domain. In the plot, the parts where $|R(z)| > 1$ are saturated to 1, but they obviously have bigger values.

- The first graph indicated the absolute value of the auto-regression constant $|R(z)|$. If its value is bigger or equal to 1, then the system will diverge, it will not be stable. As we can see, the system seems to be A-Stable because the left part of the response is not lower than 1 and from the figure the tendency also looks to be L-Stable.
- The second graph indicates the module of the relative local error,$|E(z)|$. This is the difference between the real change e^z and the autoregression constant $|R(z)|$ of the method. This time we can see that the actual error is much smaller if the real part of z is negative, so our system will perform much better than RK3 for big $|z|$ with negative real part.

$$\begin{aligned} LE_{n+1} =& y_n[e^z - R(z)] \\ =& y_n E(z) \end{aligned} \tag{52}$$

- The third and forth graphs are absolute values of the estimation of the local error of the method $\hat{LE}_{n+1}$ and the difference between the actual truncation error and the estimated one, $LE_{n+1} - \hat{LE}_{n+1}$. As we can see, these errors do not converge to 0 in the left part of the $z-$plane, this is most likely because the method of order 3 to estimate the error is not A-stable, so the estimation diverges.

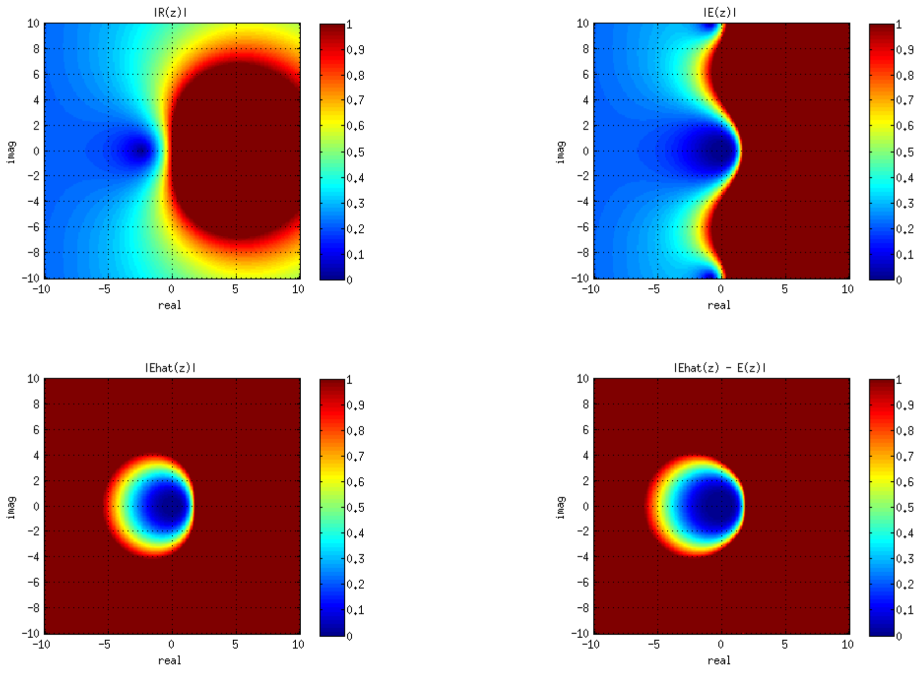


Figure 27: Stability plots of ESDIRK23

Looking at the previous graph, the method looks both A-stable and L-stable, to be sure. We can always derive the $R(z)$ function and check it, although given the previous mathematical derivation of the method is obvious that it is going to be L-stable. In this case, due to the implicit components, the function will have dividing a polynomial as well.

$$R(z) = \frac{P(z)}{Q(z)} = \frac{1 + (b_1 + b_2 - \gamma)z + (a_{21}b_2 - b_1\gamma)z^2}{(1 - \gamma z)^2} \tag{53}$$

For the system to be L-stable, the module $|R(z)|$ must tend to 0 as z tends to infinity, so the degree of P(z) has to be smaller than the degree of Q(z). This happens if $a_{21}b_2 - b_1\gamma = 0$ which was initially one of the conditions of the design of this method, so the equality holds for ESDIRK23 and the method is L-stable.

The practical application of the stability region of the ESDIRK23 is that, since it is A-stable, if the system that we solving contains exponential functions with negative exponent, then our solver is stable for any size of h. And since it is L-stable, it is also stable for any system if we choose a big enough step size. The bigger the exponent of the system, the smaller our h needs to be in order for our solver to be stable.

5.4 Implement ESDIRK23 with variable step size. Test it on the Van der Pol problem.

The implementation of this method is also performed as a simplified version of the ESDIRK.m method provided by the professor John Bagterp. In this case, we don't remove the step size control parts.
We will show the results for the test with the Van der Pol equation with $\mu = 100$. The next graphs show the solution and error estimates for both methods using a step size of $h = 0.001$ and time span from $t = 0$ to $t = 300$. The limits for the absolute and relative error have been set to $absTol = 1e-6$ and $relTol = 1e-6$.

As we can see, both systems are able to estimate the function correctly. We can see how the error estimation of the RK3 is very low for slow changing instants of the system and increases a lot in the rapidly changing ones; whereas the adaptative step ESDIRK23 maintains more or less the desired error by changing the step-size, this allows it to make a lower number of computations (bigger stepsize) in areas of the solution where the system changes slowly.

The number of function evaluations for the RK3 method is $Nf_{RK3} = 3 * Nsteps = 3 * 300.001 = 900.003$ and the number for the ESDIRK method is $Nf_{ESDIRK23} = 10.739$. In this case the ESDIRK23 method requires almost 100 times less function evaluations than the RK3. Due to the the fact that the step size will be very big in the time instants where the system varies slowly.

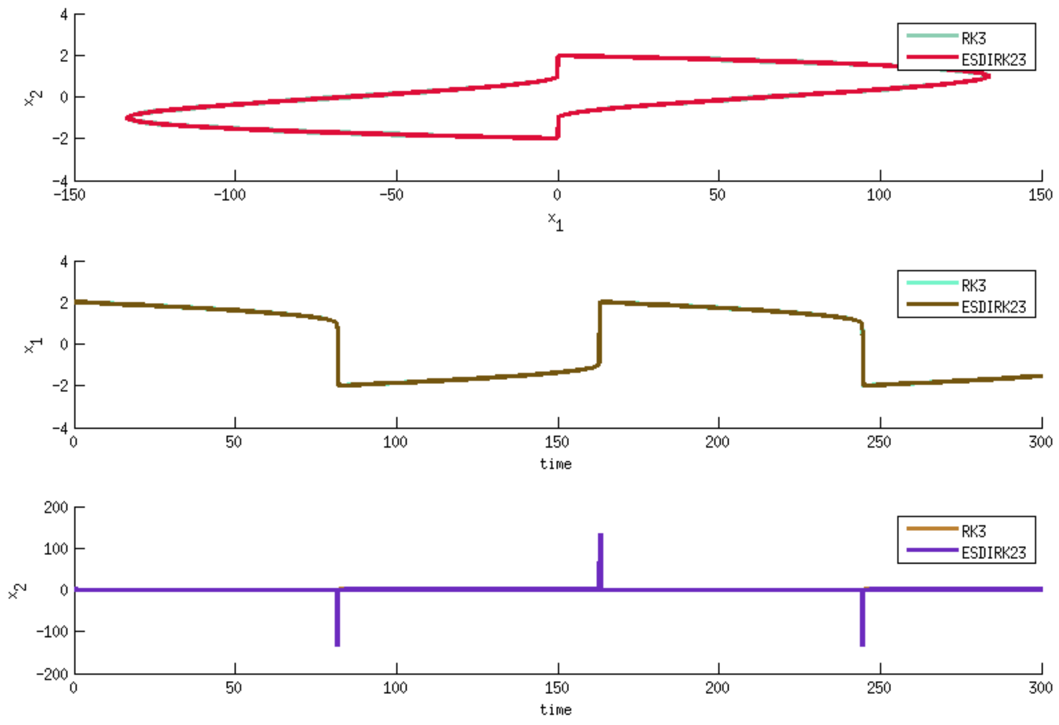


Figure 28: Solution of the solver for the methods RK3 and ESDIRK23 with adaptative steps for the Van-DerPol system with $\mu = 100$

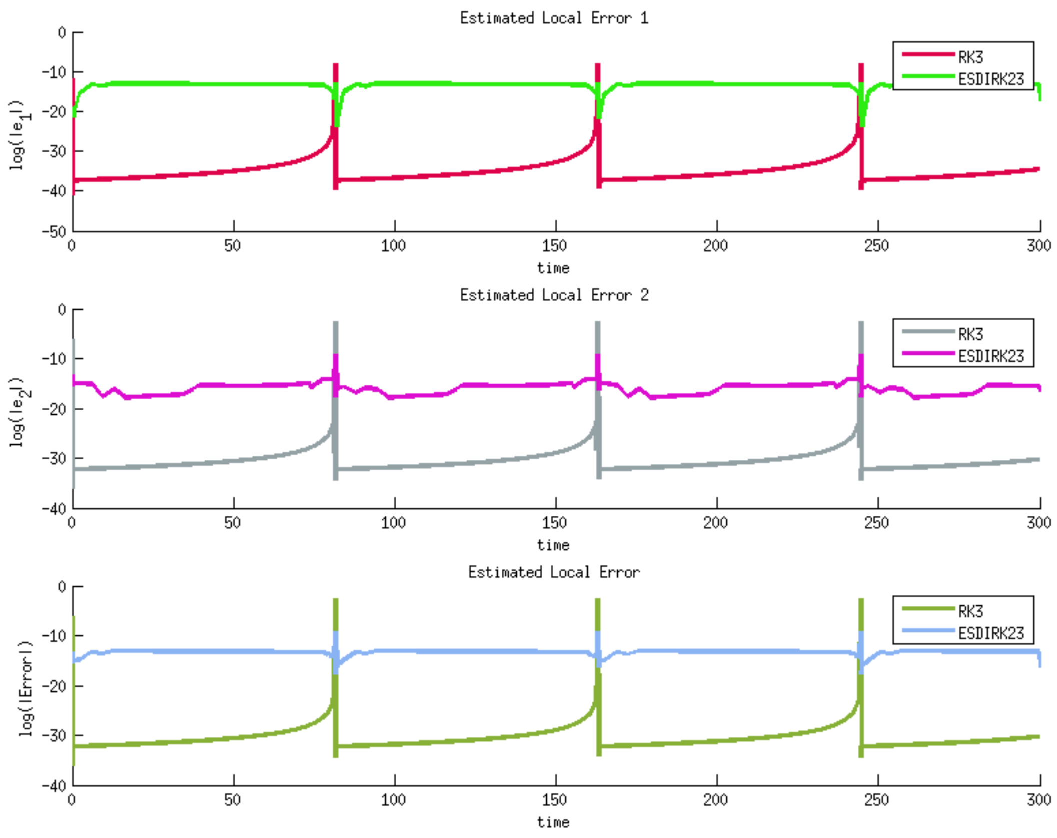


Figure 29: Estimation of the error for the methods RK3 and ESDIRK23 with adaptative steps for the VanDerPol system with $\mu = 100$

In the adaptative ESDIRK23 there were 11 failures out of 2395 steps, but none of them due to diverging of the Newton's method or slow convergence. These failures happen during the instants of high volatility of the system. This implementation seems by far the best one for stiff systems.

References

[1] John Bagterp *Lectures LectureNotesBook*

[2] Gustaf Soderlind *Automatic Control and Adaptive Time–Stepping* 2001.

A Code files

A.1 Question 1

```
%% Question 3: IVP1 global error until time t=10
close all; clear all;

% Paramters
tspan = [0 10];
N = 40;
y0 = 1;
method = 'DOPRI54';
solver = ERKSolverErrorEstimationParameters(method);
lambda = -1;
h = (tspan(2)-tspan(1))/N;

% Solvers
[tnList,ynList] = ExplicitEulers(@func,tspan,N,y0);
[tnList1,ynList1] = ImplicitEulers(@func,@Jacob,tspan,N,y0);
[tnList2,ynList2] = ImplicitTrapezoid(@func,@Jacob,tspan,N,y0);
[tnList3,ynList3] = RK4(@func,tspan,N,y0);
[tnList4,ynList4,Eout] = ...
    ERKSolverErrorEstimation(@func,tspan,y0,h,solver,lambda);

% Plotting all solvers in log scale
y = exp(lambda*tnList)*y0;
y1 = exp(lambda*tnList4)*y0;
lw = 2;
semilogy(tnList,abs(y-ynList),'LineWidth',lw)
hold on
    title('\textbf{Global error over time for} $\dot{x}(t) = -x(t)$',...
        'interpreter','latex');
    semilogy(tnList1,abs(y-ynList1),'LineWidth',lw)
    semilogy(tnList2,abs(y-ynList2),'LineWidth',lw)
    semilogy(tnList3,abs(y-ynList3),'LineWidth',lw)
    semilogy(tnList4,abs(y1-ynList4),'LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler',...
        'Trapezoidal','RK4','DOPRI54');
    set(leg,'FontSize',14);
    ylabel('log(error)','FontSize',12,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',12,'FontWeight','bold')
    grid
hold off

% Plotting Explicit vs Implicit Euler with and without log scale
figure(2)
subplot(2,1,1);  title('Global error: Explicit and Implicit Euler');
semilogy(tnList,abs(y-ynList),'LineWidth',lw)
hold on
    semilogy(tnList1,abs(y-ynList1),'LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler');
    set(leg,'FontSize',14);
    ylabel('log(Global error)','FontSize',12,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',12,'FontWeight','bold')
    grid on
hold off
subplot(2,1,2); title('Global error: without loagarithmic scale');
hold on
    plot(tnList,abs(y-ynList),'LineWidth',1)
    plot(tnList1,abs(y-ynList1),'LineWidth',1)
    leg = legend('Explicit Euler','Implicit Euler');
    set(leg,'FontSize',14);
    ylabel('Global error','FontSize',12,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',12,'FontWeight','bold')
    grid on
hold off

figure(3)
hold on
    title('\textbf{Solution for} $\dot{x}(t) = -x(t)$',...
        'interpreter','latex');
    plot(tnList,ynList,'--','LineWidth',lw)
    plot(tnList,ynList1,':','LineWidth',lw)
    plot(tnList,ynList2,'-.','LineWidth',lw)
    plot(tnList,ynList3,'--','LineWidth',lw)
    plot(tnList,ynList4,':','LineWidth',4)
    plot(tnList,y,':','LineWidth',5)
    leg = legend('Explicit Euler','Implicit Euler','Trapezoid',...
        'RK4','DOPRI54','Real');
    set(leg,'FontSize',14);
    ylabel('x(t)','FontSize',12,'FontWeight','bold')
    xlabel('t(seconds)','FontSize',12,'FontWeight','bold')
    grid on
hold off

%% Question 3: IVP2 global error until time t=10
close all; clear all;
tspan = [0 10];
N = 40;
Y0 = [0 ;1];
method = 'DOPRI54';
solver = ERKSolverErrorEstimationParameters(method);
h = (tspan(2)-tspan(1))/N;

% Solvers
[tnList,ynList] = ExplicitEulers(@IVP2,tspan,N,Y0);
[tnList1,ynList1] = ImplicitEulers(@IVP2,@jacIVP2,tspan,N,Y0);
[tnList2,ynList2] = ImplicitTrapezoid(@IVP2,@jacIVP2,tspan,N,Y0);
[tnList3,ynList3] = RK4(@IVP2,tspan,N,Y0);
[tnList4,ynList4,Eout] = ...
    ERKSolverErrorEstimation(@IVP2,tspan,Y0,h,solver);
ynList4 = ynList4';

% Analytical solution
A = [0 1;-1 0];
[m,n] = size(Y0);
timeLen = length(tnList);
Y = zeros(m,timeLen);
for i = 1:timeLen
    Y(:,i) = expm(A.*tnList(i))*Y0;
end

% Plotting
figure(1)
lw = 2;
```

```
semilogy(tnList,globError(Y,ynList),'LineWidth',lw)
hold on
    title('\textbf{Global error over time for} $\ddot{x}(t) = -x(t)$',...
        'interpreter','latex');
    semilogy(tnList,globError(Y,ynList1),'LineWidth',lw)
    semilogy(tnList,globError(Y,ynList2),'LineWidth',lw)
    semilogy(tnList,globError(Y,ynList3),'LineWidth',lw)
    semilogy(tnList,globError(Y,ynList4),'LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler',...
        'Trapezoidal','RK4','DOPRI54');
    set(leg,'FontSize',14);
    ylabel('log(Global_error)','FontSize',12,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',12,'FontWeight','bold')
    grid
hold off

% Plotting Explicit vs Implicit Euler in with and without log scale
figure(2)
subplot(2,1,1);  title('Global error: Explicit and Implicit Euler');
semilogy(tnList,globError(Y,ynList),'LineWidth',1)
hold on
    semilogy(tnList,globError(Y,ynList1),'LineWidth',1)
    leg = legend('Explicit Euler','Implicit Euler');
    set(leg,'FontSize',14);
    ylabel('log(Global error)','FontSize',12,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',12,'FontWeight','bold')
    grid on
hold off
subplot(2,1,2); title('Global error: without logarithmic scale');
hold on
    plot(tnList,globError(Y,ynList),'LineWidth',1)
    plot(tnList,globError(Y,ynList1),'LineWidth',1)
    leg = legend('Explicit Euler','Implicit Euler');
    set(leg,'FontSize',14);
    ylabel('Global error','FontSize',12,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',12,'FontWeight','bold')
    grid on
hold off

figure(3)
hold on
    title('\textbf{Solution for} $\ddot{x}(t) = -x(t)$',...
        'interpreter','latex');
    plot(tnList,ynList(1,:),'--','LineWidth',lw)
    plot(tnList,ynList1(1,:),':','LineWidth',lw)
    plot(tnList,ynList2(1,:),'-.','LineWidth',lw)
    plot(tnList,ynList3(1,:),'--','LineWidth',lw)
    plot(tnList,ynList4(1,:),':','LineWidth',4)
    plot(tnList,Y(1,:),':','LineWidth',5)
    leg = legend('Explicit Euler','Implicit Euler','Trapezoid',...
        'RK4','DOPRI54','Real');
    set(leg,'FontSize',14);
    ylabel('x(t)','FontSize',12,'FontWeight','bold')
    xlabel('t(seconds)','FontSize',12,'FontWeight','bold')
    grid on
hold off

%% Question 4: LTE error for differnet values of h IVP2
close all; clear all;
tspan = [0 10];
Y0 = [0 ;1];
```

```matlab
method = 'DOPRI54';
solver = ERKSolverErrorEstimationParameters(method);
A = [0 1;-1 0];

NList = 10:1000;
len = length(NList);
hList = zeros(1,len);
LTE = zeros(1,len);
LTE1 = zeros(1,len);
LTE2 = zeros(1,len);
LTE3 = zeros(1,len);
LTE4 = zeros(1,len);
for i = 1:len
    h = (tspan(2)-tspan(1))/NList(i);
    hList(i) = h;
    [tnList,ynList] = ExplicitEulers(@IVP2,tspan,NList(i),Y0);
    [tnList1,ynList1] = ImplicitEulers(@IVP2,@jacIVP2,tspan,NList(i),Y0);
    [tnList2,ynList2] = ImplicitTrapezoid(@IVP2,@jacIVP2,tspan,NList(i),Y0);
    [tnList3,ynList3] = RK4(@IVP2,tspan,NList(i),Y0);
    [tnList4,ynList4,Eout] = ...
        ERKSolverErrorEstimation(@IVP2,tspan,Y0,h,solver);
    ynList4 = ynList4';
    Y = expm(A.*tnList(2))*Y0;

    LTE(i) = norm(ynList(:,2)-Y);
    LTE1(i) = norm(ynList1(:,2)-Y);
    LTE2(i) = norm(ynList2(:,2)-Y);
    LTE3(i) = norm(ynList3(:,2)-Y);
    LTE4(i) = norm(ynList4(:,2)-Y);
end
%%
m1= (log(LTE(100))-log(LTE(200)))/(log(hList(100))-log(hList(200)));
m2= (log(LTE1(100))-log(LTE1(200)))/(log(hList(100))-log(hList(200)));
m3= (log(LTE2(100))-log(LTE2(200)))/(log(hList(100))-log(hList(200)));
m4= (log(LTE3(100))-log(LTE3(200)))/(log(hList(100))-log(hList(200)));
m5= (log(LTE4(100))-log(LTE4(200)))/(log(hList(100))-log(hList(200)));
%%
close all;
lw = 2;
figure(1)
loglog(hList,LTE,'LineWidth',lw)
hold on
    title('\textbf{LTE vs step size for} $\ddot{x}(t) = -x(t)$',...
        'interpreter','latex','FontSize',14);

    loglog(hList,LTE1,'--','LineWidth',lw)
    loglog(hList,LTE2,'LineWidth',lw)
    loglog(hList,LTE3,'LineWidth',lw)
    loglog(hList,LTE4,'LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler',...
        'Trapezoidal','RK4','DOPRI54');
    set(leg,'FontSize',14);
    ylabel('log(LTE)','FontSize',12,'FontWeight','bold')
    xlabel('log(h)','FontSize',12,'FontWeight','bold')
    grid on
hold off

% Plotting Explicit vs Implicit Euler in with and without log scale
figure(2)
subplot(2,1,1);  title('LTE: Explicit and Implicit Euler');
loglog(hList,LTE,'LineWidth',lw)
```

```
hold on
    loglog(hList,LTE1,'--','LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler');
    set(leg,'FontSize',14);
    ylabel('log(LTE)','FontSize',12,'FontWeight','bold')
    xlabel('log(h)','FontSize',12,'FontWeight','bold')
    grid on
hold off
subplot(2,1,2); title('LTE: without logarithmic scale');
hold on
    plot(hList,LTE,'LineWidth',lw)
    plot(hList,LTE1,'--','LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler');
    set(leg,'FontSize',14);
    ylabel('LTE error','FontSize',12,'FontWeight','bold')
    xlabel('h','FontSize',12,'FontWeight','bold')
    grid on
hold off

%% Question 4: LTE error for differnet values of h IVP1
close all; clear all;
tspan = [0 10];
y0 = 1;
method = 'DOPRI54';
solver = ERKSolverErrorEstimationParameters(method);
lambda = -1;

NList = 10:1000;
len = length(NList);
hList = zeros(1,len);
LTE = zeros(1,len);
LTE1 = zeros(1,len);
LTE2 = zeros(1,len);
LTE3 = zeros(1,len);
LTE4 = zeros(1,len);
for i = 1:len
    h = (tspan(2)-tspan(1))/NList(i);
    hList(i) = h;
    [tnList,ynList] = ExplicitEulers(@func,tspan,NList(i),y0);
    [tnList1,ynList1] = ImplicitEulers(@func,@Jacob,tspan,NList(i),y0);
    [tnList2,ynList2] = ImplicitTrapezoid(@func,@Jacob,tspan,NList(i),y0);
    [tnList3,ynList3] = RK4(@func,tspan,NList(i),y0);
    [tnList4,ynList4,Eout] = ...
    ERKSolverErrorEstimation(@func,tspan,y0,h,solver,lambda);
    ynList4 = ynList4';

    y = exp(lambda*tnList(2))*y0;

    LTE(i) = norm(ynList(:,2)-y);
    LTE1(i) = norm(ynList1(:,2)-y);
    LTE2(i) = norm(ynList2(:,2)-y);
    LTE3(i) = norm(ynList3(:,2)-y);
    LTE4(i) = norm(ynList4(:,2)-y);
end
%%
m1= (log(LTE(100))-log(LTE(200)))/(log(hList(100))-log(hList(200)));
m2= (log(LTE1(100))-log(LTE1(200)))/(log(hList(100))-log(hList(200)));
m3= (log(LTE2(100))-log(LTE2(200)))/(log(hList(100))-log(hList(200)));
m4= (log(LTE3(100))-log(LTE3(200)))/(log(hList(100))-log(hList(200)));
m5= (log(LTE4(100))-log(LTE4(200)))/(log(hList(100))-log(hList(200)));
%%
```

```
close all;
lw = 2;
loglog(hList,LTE,'LineWidth',lw)
hold on
    title('\textbf{LTE vs step size for} $\dot{x}(t) = -x(t)$',...
        'interpreter','latex','FontSize',14);

    loglog(hList,LTE1,'--','LineWidth',lw)
    loglog(hList,LTE2,'LineWidth',lw)
    loglog(hList,LTE3,'LineWidth',lw)
    loglog(hList,LTE4,'LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler',...
        'Trapezoidal','RK4','DOPRI54');
    set(leg,'FontSize',14);
    ylabel('log(LTE)','FontSize',12,'FontWeight','bold')
    xlabel('log(h)','FontSize',12,'FontWeight','bold')
    grid on
hold off

% Plotting Explicit vs Implicit Euler in with and without log scale
figure()
subplot(2,1,1);
loglog(hList,LTE,'LineWidth',lw)
hold on
    title('LTE: Explicit and Implicit Euler')
    loglog(hList,LTE1,'--','LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler');
    set(leg,'FontSize',14);
    ylabel('log(LTE)','FontSize',12,'FontWeight','bold')
    xlabel('log(h)','FontSize',12,'FontWeight','bold')
    grid on
hold off
subplot(2,1,2); title('LTE: without logarithmic scale');
hold on
    plot(hList,LTE,'LineWidth',lw)
    plot(hList,LTE1,'--','LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler');
    set(leg,'FontSize',14);
    ylabel('LTE error','FontSize',12,'FontWeight','bold')
    xlabel('h','FontSize',12,'FontWeight','bold')
    grid on
hold off

% Calculate slopes for order checking
m1= (log(LTE(10))-log(LTE(30)))/(log(hList(10))-log(hList(30)));
m2= (log(LTE1(10))-log(LTE1(30)))/(log(hList(10))-log(hList(30)));
m3= (log(LTE2(10))-log(LTE2(30)))/(log(hList(10))-log(hList(30)));
m4= (log(LTE3(10))-log(LTE3(30)))/(log(hList(10))-log(hList(30)));
m5= (log(LTE4(10))-log(LTE4(30)))/(log(hList(10))-log(hList(30)));
 %% Question 5: error estimation for IVP1
close all; clear all;
tspan = [0 10];
y0 = 1;
method = 'DOPRI54';
solver = ERKSolverErrorEstimationParameters(method);
lambda = -1;

NList = 10:1000;
len = length(NList);
hList = zeros(1,len);
LTEest1 = zeros(1,len);
```

```
LTEreal1 = zeros(1,len);
LTEest2 = zeros(1,len);
LTEreal2 = zeros(1,len);
LTEest3 = zeros(1,len);
LTEreal3 = zeros(1,len);
LTEest4 = zeros(1,len);
LTEreal4 = zeros(1,len);
LTEest5 = zeros(1,len);
LTEreal5 = zeros(1,len);
for i = 1:len
    h = (tspan(2)-tspan(1))/NList(i);
    hList(i) = h;
    [tnList1,ynList1] = ExplicitEulers(@func,tspan,NList(i),y0);
    [tnList11,ynList11] = ExplicitEulers(@func,tspan,2*NList(i),y0);
    [tnList2,ynList2] = ImplicitEulers(@func,@Jacob,tspan,NList(i),y0);
    [tnList22,ynList22] = ImplicitEulers(@func,@Jacob,tspan,2*NList(i),y0);
    [tnList3,ynList3] = ImplicitTrapezoid(@func,@Jacob,tspan,NList(i),y0);
    [tnList33,ynList33] = ImplicitTrapezoid(@func,@Jacob,tspan,2*NList(i),y0);
    [tnList4,ynList4] = RK4(@func,tspan,NList(i),y0);
    [tnList44,ynList44] = RK4(@func,tspan,2*NList(i),y0);

    y = exp(lambda*tnList1(2))*y0;
    LTEest1(i) = 2*norm(ynList1(2)-ynList11(3));
    LTEreal1(i) = norm(ynList1(2)-y);
    LTEest2(i) = 2*norm(ynList2(2)-ynList22(3));
    LTEreal2(i) = norm(ynList2(2)-y);
    LTEest3(i) = (4/3)*norm(ynList3(2)-ynList33(3));
    LTEreal3(i) = norm(ynList3(2)-y);
    LTEest4(i) = (16/15)*norm(ynList4(2)-ynList44(3));
    LTEreal4(i) = norm(ynList4(2)-y);
end
%%
figure(1)
loglog(hList,LTEest1,':','LineWidth',3)
hold on
    loglog(hList,LTEreal1,'--','LineWidth',2)
    loglog(hList,LTEest2,':','LineWidth',2)
    loglog(hList,LTEreal2,'--','LineWidth',2)
    loglog(hList,LTEest3,':','LineWidth',2)
    loglog(hList,LTEreal3,'--','LineWidth',2)
    loglog(hList,LTEest4,':','LineWidth',2)
    loglog(hList,LTEreal4,'--','LineWidth',3)
    leg = legend('Error estimation ExEuler','Real error ExEuler',...
        'Error estimation Trapezoid','Real error Trapezoid',...
        'Error estimation RK4','Real error RK4');
    set(leg,'FontSize',12);
    ylabel('log(LTE)','FontSize',12,'FontWeight','bold')
    xlabel('log(h)','FontSize',12,'FontWeight','bold')
    grid on
hold off

%% Question 5: error estimation for IVP2
close all; clear all;
tspan = [0 10];
Y0 = [0 ;1];
method = 'DOPRI54';
solver = ERKSolverErrorEstimationParameters(method);
A = [0 1;-1 0];

NList = 10:1000;
len = length(NList);
```

```
hList = zeros(1,len);
LTEest1 = zeros(1,len);
LTEreal1 = zeros(1,len);
LTEest2 = zeros(1,len);
LTEreal2 = zeros(1,len);
LTEest3 = zeros(1,len);
LTEreal3 = zeros(1,len);
LTEest4 = zeros(1,len);
LTEreal4 = zeros(1,len);
LTEest5 = zeros(1,len);
LTEreal5 = zeros(1,len);
LTEembed5 = zeros(1,len);
for i = 1:len
    h = (tspan(2)-tspan(1))/NList(i);
    hList(i) = h;
    [tnList1,ynList1] = ExplicitEulers(@IVP2,tspan,NList(i),Y0);
    [tnList11,ynList11] = ExplicitEulers(@IVP2,tspan,2*NList(i),Y0);
    [tnList2,ynList2] = ImplicitEulers(@IVP2,@jacIVP2,tspan,NList(i),Y0);
    [tnList22,ynList22] = ImplicitEulers(@IVP2,@jacIVP2,tspan,2*NList(i),Y0);
    [tnList3,ynList3] = ImplicitTrapezoid(@IVP2,@jacIVP2,tspan,NList(i),Y0);
    [tnList33,ynList33] = ImplicitTrapezoid(@IVP2,@jacIVP2,tspan,2*NList(i),Y0);
    [tnList4,ynList4] = RK4(@IVP2,tspan,NList(i),Y0);
    [tnList44,ynList44] = RK4(@IVP2,tspan,2*NList(i),Y0);
    Y = expm(A.*tnList1(2))*Y0;
    [tnList5,ynList5,Eout] = ...
        ERKSolverErrorEstimation(@IVP2,tspan,Y0,h,solver);
    ynList5 = ynList5';
    [tnList55,ynList55,Eout1] = ...
        ERKSolverErrorEstimation(@IVP2,tspan,Y0,h/2,solver);
    ynList55 = ynList55';

    LTEest1(i) = 2*norm(ynList1(:,2)-ynList11(:,3));
    LTEreal1(i) = norm(ynList1(:,2)-Y);
    LTEest2(i) = 2*norm(ynList2(:,2)-ynList22(:,3));
    LTEreal2(i) = norm(ynList2(:,2)-Y);
    LTEest3(i) = (4/3)*norm(ynList3(:,2)-ynList33(:,3));
    LTEreal3(i) = norm(ynList3(:,2)-Y);
    LTEest4(i) = (16/15)*norm(ynList4(:,2)-ynList44(:,3));
    LTEreal4(i) = norm(ynList4(:,2)-Y);
    LTEest5(i) = (32/31)*norm(ynList5(:,2)-ynList55(:,3));
    LTEreal5(i) = norm(ynList5(:,2)-Y);
    LTEembed5(i) = norm(Eout(2,:));
end
%%
close all;
figure(1)
loglog(hList,LTEest1,':','LineWidth',3)
hold on
    title('Error estimaton by step-doubling')
    loglog(hList,LTEreal1,'--','LineWidth',2)
    loglog(hList,LTEest2,':','LineWidth',2)
    loglog(hList,LTEreal2,'--','LineWidth',2)
    loglog(hList,LTEest3,':','LineWidth',2)
    loglog(hList,LTEreal3,'--','LineWidth',2)
    loglog(hList,LTEest4,':','LineWidth',2)
    loglog(hList,LTEreal4,'--','LineWidth',3)
    leg = legend('Error estimation ExEuler','Real error ExEuler',...
        'Error estimation ImpEuler','Real error ImEuler',...
        'Error estimation Trapezoid','Real error Trapezoid',...
        'Error estimation RK4','Real error RK4');
    set(leg,'FontSize',12);
    ylabel('log(LTE)','FontSize',12,'FontWeight','bold')
    xlabel('log(h)','FontSize',12,'FontWeight','bold')
    grid on
hold off

figure(2)
loglog(hList,LTEreal5,':','LineWidth',4)
hold on
    title('DOPRI error estimation')
    loglog(hList,LTEembed5,'--','LineWidth',2)
    loglog(hList,LTEest5,'-.','LineWidth',2)
    leg = legend('Real error','Embbeded estimation',...
        'Step-doubling Estimation');
    set(leg,'FontSize',12);
    ylabel('log(LTE)','FontSize',12,'FontWeight','bold')
    xlabel('log(h)','FontSize',12,'FontWeight','bold')
    grid on
hold off

figure(3)
loglog(hList,LTEest1,':','LineWidth',2)
hold on
    title('Explicit and Implicit Euler error estimation')
    loglog(hList,LTEreal1,'--','LineWidth',2)
    loglog(hList,LTEest2,':','LineWidth',2)
    loglog(hList,LTEreal2,'--','LineWidth',2)
    leg = legend('Error estimation ExEuler','Real error ExEuler',...
        'Error estimation ImpEuler','Real error ImEuler');
    set(leg,'FontSize',12);
    ylabel('log(LTE)','FontSize',12,'FontWeight','bold')
    xlabel('log(h)','FontSize',12,'FontWeight','bold')
    grid on
hold off

function [tnList,ynList] = ExplicitEulers(func,tspan,N,Y0)
%
% This function solves a general first-order Initial Value Problem
% of the form
%                u? = f(u,t),  u(tstart) = eta
%
% using Euler?s Method in n steps (constant step size).
%
% INPUT:
%    func  : a function handle to function f(u,t)
%    tspan : a 1x2 array of the form [tstart tend]
%    N     : total number of steps in tspan
%    y0   : initialvalue(s)
%    param : parameters to be passed to func
%
sizeY = size(Y0);
Ndim = sizeY(1);
sizeTspan = size(tspan);
Ninit = sizeTspan(2);  % We can be given only the end time, then the begining is 0

ynList = zeros(Ndim,N+1);
tnList = zeros(1,N+1);

%% Initialization
if (Ninit == 1)
    tbegin = 0;
    tend = tspan;
```

```
elseif(Ninit == 2) % [tbegin tend]
    tbegin = tspan(1,1);
    tend = tspan(1,2);
end
dt = (tend - tbegin)/N;

tnList(1) = tbegin;
ynList(:,1) = Y0;

%% Loop

for k = 1:N
    f = feval(func,tnList(k),ynList(:,k));
    ynList(:,k+1) = ynList(:,k)+dt*f;
    tnList(k+1) = tnList(k)+dt;
end

 function [tnList,ynList] = ImplicitEulers(func,Jacob,tspan,N,Y0)
%
% This function solves a general first-order Initial Value Problem
% of the form
%                 Ydot = F(y,t),  y(tbegin) = Y0
%
% using Implicit Euler Method with adaptive time step
%
% INPUT:
%    func   : a function handle to function F(Y,t)
%    Jacob  : a function handle to function dF(Y,t)/dY
%    tspan  : a 1x2 array of the form [tbegin tend]
%    N      : total number of steps in tspan
%    Y0     : initialvalue(s)s
%

sizeY = size(Y0);
Ndim = sizeY(1);
sizeTspan = size(tspan);
Ninit = sizeTspan(2);

%% Initialization
if (Ninit == 1)
    tbegin = 0;
    tend = tspan;
elseif(Ninit == 2) % [tbegin tend]
    tbegin = tspan(1,1);
    tend = tspan(1,2);
end
h = (tend - tbegin)/N;

ynList = zeros(Ndim,N+1);
tnList = zeros(1,N+1);
tol = 10e-5; % Newtons method tolerance
ynList(:,1)=Y0;

%% Loop
for k = 1:N
    f = feval(func,tnList(k),ynList(:,k));
    y_guess = ynList(:,k)+h*f; %Forwards euler
    ynList(:,k+1) = newtonODE(func,Jacob,y_guess,tol,tnList(k)+h,ynList(:,k),h);
    tnList(:,k+1) = tnList(:,k)+h;
end

function [tnList,ynList] = ImplicitTrapezoid(func,Jacob,tspan,N,Y0)
```

```
%
% This function solves a general first-order Initial Value Problem
% of the form
%                 u? = f(u,t),  u(tstart) = eta
%
% using Euler?s Method in n steps (constant step size).
%
% INPUT:
%    func  : a function handle to function f(u,t)
%    tspan : a 1x2 array of the form [tstart tend]
%    N     : total number of steps in tspan
%    y0    : initialvalue(s)
%    param : parameters to be passed to func
%
sizeY = size(Y0);
Ndim = sizeY(1);

sizeTspan = size(tspan);
Ninit = sizeTspan(2);  % We can be given only the end time, then the begining is 0

ynList = zeros(Ndim,N+1);
tnList = zeros(1,N+1);

%% Initialization
if (Ninit == 1)
    tbegin = 0;
    tend = tspan;
elseif(Ninit == 2) % [tbegin tend]
    tbegin = tspan(1,1);
    tend = tspan(1,2);
end
h = (tend - tbegin)/N;

tol = 10e-5; % Newtons method tolerance
tnList(1) = tbegin;
ynList(:,1) = Y0;

%% Loop
for k = 1:N
    f = feval(func,tnList(k),ynList(:,k));

    % Calculate the next fn+1 computed by the Explicit E
    yn_guess = ynList(:,k) + h*f;
    yn = newtonODE(func,Jacob,yn_guess,tol,tnList(k)+h,ynList(:,k),h);
    fn = feval(func,tnList(k) + h ,yn);

    ynList(:,k+1) = ynList(:,k)+ (h/2)*(fn + f);
    tnList(k+1) = tnList(k)+h;
end

function [tnList,ynList] = ExplicitTrapezoid(func,tspan,N,Y0)
%
% This function solves a general first-order Initial Value Problem
% of the form
%                 u? = f(u,t),  u(tstart) = eta
%
% using Euler?s Method in n steps (constant step size).
%
% INPUT:
%    func  : a function handle to function f(u,t)
%    tspan : a 1x2 array of the form [tstart tend]
%    N     : total number of steps in tspan
%    y0   : initialvalue(s)
%    param : parameters to be passed to func
%
sizeY = size(Y0);
Ndim = sizeY(1);

sizeTspan = size(tspan);
Ninit = sizeTspan(2);  % We can be given only the end time, then the begining is 0

ynList = zeros(Ndim,N+1);
tnList = zeros(1,N+1);

%% Initialization
if (Ninit == 1)
    tbegin = 0;
    tend = tspan;
elseif(Ninit == 2) % [tbegin tend]
    tbegin = tspan(1,1);
    tend = tspan(1,2);
end
dt = (tend - tbegin)/N;

tnList(1) = tbegin;
ynList(:,1) = Y0;

%% Loop
for k = 1:N
    f = feval(func,tnList(k),ynList(:,k));

    % Calculate the next fn+1 computed by the Explicit E
    ynExplicitEuler = ynList(:,k) + dt*f;
    fnExplicitEuler = feval(func,tnList(k) + dt ,ynExplicitEuler);

    ynList(:,k+1) = ynList(:,k)+ (dt/2)*(fnExplicitEuler + f);
    tnList(k+1) = tnList(k)+dt;
end

function [error] = globError(Y,ynList)
[m,timeLen] = size(ynList);
error = zeros(1,timeLen);
for i = 1:timeLen
    error(i) = norm(Y(:,i)-ynList(:,i));
end
end

function [ yn1_n1 ] = newtonODE(func,Jacob,y0,tol,tn1,yn,h)
%Newton root finding algorithm
%   Fibd the roots of F(yn1) = yn1 - h*f(yn1) - yn
%   where yn1 is the unknown variable
% INPUT:
%    func  : a function handle to function f(y,t)
%    Jacob : jacobian of f(y,t)
%    y0   :  intial guess for the root i.e yn1_0
%    param : parameters to be passed to func
%    tol : tolerance for the solution
%    yn,tn1,h; are constant paramters of the fucntion F(yn1)
% OUTPUT:
%    yn1_n1  : estimated root

yn1_n = y0;
maxIt = 100;
```

```
err = 1000;
it = 0;
I = eye(length(y0));

while((it < maxIt) && (norm(err,'inf') > tol))
    it = it +1;
    fprime = I-h*feval(Jacob,tn1,yn1_n);
    f = yn1_n-h*feval(func,tn1,yn1_n)-yn;
    yn1_n1 = yn1_n - fprime\f;
    f = yn1_n1-h*feval(func,tn1,yn1_n1)-yn;
    err = f;
    yn1_n = yn1_n1;
end
end
```

A.2 Question 2

```
%% Question 1: mu = 3
close all; clear all;

% Paramters
tspan = [0 100];
N = 100000;
y0 = [1;1];
method = 'DOPRI54';
solver = ERKSolverErrorEstimationParameters(method);
h = (tspan(2)-tspan(1))/N

% Solvers
[tnList1,ynList1] = ExplicitEulers(@VanDerPol,tspan,N,y0);
[tnList2,ynList2] = ImplicitEulers(@VanDerPol,@jacVanDerPol,tspan,N,y0);
[tnList3,ynList3] = ImplicitTrapezoid(@VanDerPol,@jacVanDerPol,tspan,N,y0);
[tnList4,ynList4] = RK4(@VanDerPol,tspan,N,y0);
[tnList5,ynList5,Eout5] = ...
    ERKSolverErrorEstimation(@VanDerPol,tspan,y0,h,solver);
ynList5 = ynList5';

% Double of steps for making error estimation
[tnList11,ynList11] = ExplicitEulers(@VanDerPol,tspan,N*2,y0);
[tnList22,ynList22] = ImplicitEulers(@VanDerPol,@jacVanDerPol,tspan,N*2,y0);
[tnList33,ynList33] = ImplicitTrapezoid(@VanDerPol,@jacVanDerPol,tspan,N*2,y0);
[tnList44,ynList44] = RK4(@VanDerPol,tspan,N*2,y0);
[tnList55,ynList55,Eout55] = ...
    ERKSolverErrorEstimation(@VanDerPol,tspan,y0,h/2,solver);
ynList55 = ynList55';

len = length(ynList1);
LTEest1 = zeros(1,len);
LTEest2 = zeros(1,len);
LTEest3 = zeros(1,len);
LTEest4 = zeros(1,len);
LTEest5 = zeros(1,len);
LTEembed5 = zeros(1,len);
for i = 2:len
    LTEest1(i) = 2*norm(ynList1(:,i)-ynList11(:,i+i-1));
    LTEest2(i) = 2*norm(ynList2(:,i)-ynList22(:,i+i-1));
    LTEest3(i) = (4/3)*norm(ynList3(:,i)-ynList33(:,i+i-1));
    LTEest4(i) = (16/15)*norm(ynList4(:,i)-ynList44(:,i+i-1));
    LTEest5(i) = (32/31)*norm(ynList5(:,i)-ynList55(:,i+i-1));
    LTEembed5(i) = norm(Eout5(2,:));
end
```

```
%% Plotting question 1
lw = 2;
figure(1)
hold on
    title('Van Der Pol mu = 100');
    plot(ynList1(1,:),ynList1(2,:),'-.','LineWidth',lw)
    plot(ynList2(1,:),ynList2(2,:),':','LineWidth',lw)
    plot(ynList3(1,:),ynList3(2,:),'--','LineWidth',lw)
    plot(ynList4(1,:),ynList4(2,:),'--','LineWidth',lw+1)
    plot(ynList5(1,:),ynList5(2,:),':','LineWidth',lw+2)
    leg = legend('Explicit Euler','Implicit euler',...
        'Trapezoidal','RK4','DOPRI54');
    set(leg,'FontSize',12);
    ylabel('y(t)','FontSize',12,'FontWeight','bold')
    xlabel('x(t)','FontSize',12,'FontWeight','bold')
    grid on
hold off

figure(2)
semilogy(tnList1,LTEest1,'LineWidth',lw)
hold on
    title('Estimated error for mu = 100');
    semilogy(tnList1,LTEest2,'LineWidth',lw)
    semilogy(tnList1,LTEest3,'LineWidth',lw)
    semilogy(tnList1,LTEest4,'LineWidth',lw)
    semilogy(tnList1,LTEest5,'LineWidth',lw)
    leg = legend('Explicit Euler','Implicit euler',...
        'Trapezoidal','RK4','DOPRI54');
    set(leg,'FontSize',12);
    ylabel('log(LTEest)','FontSize',12,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',12,'FontWeight','bold')
    grid on
hold off
%%
NList = 10:1000;
len = length(NList);
hList = zeros(1,len);
LTEest1 = zeros(1,len);
LTEest2 = zeros(1,len);
LTEest3 = zeros(1,len);
LTEest4 = zeros(1,len);
LTEest5 = zeros(1,len);
LTEembed5 = zeros(1,len);
for i = 1:len
    h = (tspan(2)-tspan(1))/NList(i);
    hList(i) = h;
    [tnList1,ynList1] = ExplicitEulers(@VanDerPol,tspan,NList(i),y0);
    [tnList11,ynList11] = ExplicitEulers(@VanDerPol,tspan,NList(i),y0);
    [tnList2,ynList2] = ImplicitEulers(@VanDerPol,@jacVanDerPol,tspan,NList(i),y0);
    [tnList22,ynList22] = ImplicitEulers(@VanDerPol,@jacVanDerPol,tspan,NList(i),y0);
    [tnList3,ynList3] = ExplicitTrapezoid(@VanDerPol,tspan,NList(i),y0);
    [tnList33,ynList33] = ExplicitTrapezoid(@VanDerPol,tspan,NList(i),y0);
    [tnList4,ynList4] = RK4(@VanDerPol,tspan,NList(i),y0);
    [tnList44,ynList44] = RK4(@VanDerPol,tspan,NNList(i),y0);
    [tnList5,ynList5,Eout1] = ...
    ERKSolverErrorEstimation(@VanDerPol,tspan,y0,h,solver,lambda);
    [tnList55,ynList55,Eout2] = ...
    ERKSolverErrorEstimation(@VanDerPol,tspan,y0,h,solver,lambda);

    LTEest1(i) = 2*norm(ynList1(:,2)-ynList11(:,3));
    LTEest2(i) = 2*norm(ynList2(:,2)-ynList22(:,3));
    LTEest3(i) = (4/3)*norm(ynList3(:,2)-ynList33(:,3));
    LTEest4(i) = (16/15)*norm(ynList4(:,2)-ynList44(:,3));
    LTEest5(i) = (32/31)*norm(ynList5(:,2)-ynList55(:,3));
    LTEembed5(i) = norm(Eout(2,:));
end

function [ jac ] = jacVanDerPol( t, Y, args )
%VANDELPOL Summary of this function goes here
%   Detailed explanation goes here
    mu = 100;
    x = Y(1);
    y = Y(2);
    jac = zeros(length(Y));
    % Syntax: xdot = PreyPredator(t,x,a,b)
    jac(1,1) = 0;
    jac(1,2) = 1;
    jac(2,1) = -2*mu*x*y -1;
    jac(2,1) = mu*(1-x^2);
end

function [ yn1_n1 ] = newtonODE(func,Jacob,y0,tol,tn1,yn,h)
%Newton root finding algorithm
%   Fibd the roots of F(yn1) = yn1 - h*f(yn1) - yn
%   where yn1 is the unknown variable
% INPUT:
%    func  : a function handle to function f(y,t)
%    Jacob : jacobian of f(y,t)
%    y0    :  intial guess for the root i.e yn1_0
%    param : parameters to be passed to func
%    tol : tolerance for the solution
%    yn,tn1,h; are constant paramters of the fucntion F(yn1)
% OUTPUT:
%    yn1_n1   : estimated root

yn1_n = y0;
maxIt = 100;
err = 1000;
it = 0;
I = eye(length(y0));

while((it < maxIt) && (norm(err,'inf') > tol))
    it = it +1;
    fprime = I-h*feval(Jacob,tn1,yn1_n);
    f = yn1_n-h*feval(func,tn1,yn1_n)-yn;
    yn1_n1 = yn1_n - fprime\f;
    f = yn1_n1-h*feval(func,tn1,yn1_n1)-yn;
    err = f;
    yn1_n = yn1_n1;
end
end
```

A.3 Question 3

```
%% Explicit eulers
close all; clear all;

lw = 3;


%% Set solver parameters and simulation scenario
% ==========================================================================
method = 'DOPRI54';
```

```
solver = ERKSolverErrorEstimationParameters(method);
lambda = -1;
x0 = 1;
tspan = [0 10];
h = 0.1;

%% Our RK3 method

%% Compute numerical solution, analytical solution, and errors
% =========================================================================
[Tout,Xout,Eout] = RK3(@func,tspan,x0,h,lambda);
X = x0*exp(lambda*Tout);  % Real value
Eglobal = X-Xout;
EglobalK = Eglobal; % Normalized by the signal

Elocal = zeros(size(Eout));
ElocalK = zeros(size(Eout));  % Constant of error

%% Compute the actual local error

for i=2:length(Tout)
    Xlocal = Xout(i-1)*exp(lambda*(Tout(i)-Tout(i-1)));
    Elocal(i) = Xout(i) - Xlocal;
    ElocalK(i) = Elocal(i) / Xout(i-1);
end
% The first local error will be 0, becaute it is the initial conditions, so
% we get
Elocal(1) = Elocal(2);
Eout(1) = Eout(2);
%% Plotting
figure()

subplot(1,3,1);
title('Real and Simulated Functions', 'fontSize',20,'fontWeight','Bold');
hold on
    plot(Tout, Xout, 'color',rand(1,3), 'LineWidth',lw)
    plot(Tout, x0*exp(lambda*Tout), 'color',rand(1,3),'LineWidth',lw)  %e^(lambda*x)*x0
    legend('Simulated', 'Real')

    ylabel('U','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')

hold off

subplot(1,3,2); title('Actual and Estimated Local Error');
hold on
    plot(Tout, log(abs(Eout)), 'color',rand(1,3), 'LineWidth',lw)
    plot(Tout, log(abs(Elocal)), 'color',rand(1,3), 'LineWidth',lw)
    plot(Tout, log(abs(ElocalK)), 'color',rand(1,3), 'LineWidth',lw)

    legend('Estimated Local Error', 'Actual Local Error', 'Normalized Local Error')
    ylabel('log(|Error|)','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')

hold off

subplot(1,3,3); title('Global error');
hold on
    plot(Tout, abs(x0*exp(lambda*Tout) - Xout), 'color',rand(1,3), 'LineWidth',lw)
    legend('Global Error')
    ylabel('|Error|','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')

hold off


%% Study with the stepsize

N = 30;
tspan = [0 10];
hs = (1:N)/100;

LEs = zeros(N,1);
LTEs = zeros(N,1);
for ih = 1:N
    h = hs(1,ih);
    [Tout,Xout,Eout] = RK3(@func,tspan,x0,h,lambda);
    X = x0*exp(lambda*Tout);  % Real value
    Eglobal = X-Xout;

    Elocal = zeros(size(Eout));
    for i=2:length(Tout)
        Xlocal = Xout(i-1)*exp(lambda*(Tout(i)-Tout(i-1)));
        Elocal(i) = Xout(i) - Xlocal;
    end
    LEs(ih) = Elocal(2);
    LTEs(ih) = Eout(2);
end

%% Compute the actual local error
figure()

hsLog = log(hs);
LELog = log(abs(LEs));
LTELog = log(abs(LTEs));

slopeLE = (LELog(4) - LELog(3))/(hsLog(4) - hsLog(3));
slopeLTE = (LTELog(4) - LTELog(3))/(hsLog(4) - hsLog(3));
hold on
    title('Relation between local error and h','FontSize',12,'FontWeight','bold')
    plot(hsLog, LELog, 'color',rand(1,3), 'LineWidth',lw)
    plot(hsLog, LTELog, 'color',rand(1,3), 'LineWidth',lw)
    legend(strcat('Real. Slope = ',num2str(slopeLE)), strcat('Estimation. Slope = ',num2str(slopeLTE
    ylabel('log(|Error|)','FontSize',12,'FontWeight','bold')
    xlabel('log(h)','FontSize',12,'FontWeight','bold')
hold off


h = 0.1;
lambda = -1;

ts = 0:h:10;
N = size(ts);
N = N(2);
% Global error
Real = zeros(N,1);
Est = zeros(N,1);

for i = 1:N
    Real(i) = exp(lambda*0)*exp(lambda*ts(i));
    Est(i) = exp(lambda*0)*power (1 - h, i-1);
end
```

```
figure()
hold on
    plot(ts, abs(Real - Est))
    plot(ts, abs( Est))
    plot(ts, abs(Real))
    legend('Error', 'Estimation', 'Real')
 %   plot(ts, abs(Real - Est)./Est)
hold off

%% Explicit eulers
close all; clear all;

lw = 3;

alpha = -5:0.01:5;
beta = -5:0.01:5;

A = [0 0 0;
    1/2 0 0;
    -1 2 0];
b = [1/6; 2/3; 1/6];
c = [0; 1; 1/2];
d = [1/12; -1/6; 1/12];
s = 3;

AT = A.';
nreal = length(alpha);
nimag = length(beta);
I = eye(size(A));
e = ones(size(A,1),1);

for kreal = 1:nreal
    for kimag = 1:nimag
        z = alpha(kreal) + 1i*beta(kimag);
        tmp = (I-z*A)\e;
        R = 1 + z*b.'*tmp;

        Ehat = z*d.'*tmp; % This is the estimated local error
        f = exp(z); % This is the real variation
        E = R-f;    % This is the real local error
        EhatmE = Ehat-E; % This is the difference between the estimated error and the real error.

        absR(kimag,kreal) = abs(R);
        absEhatmE(kimag,kreal) = abs(EhatmE);
        absEhat(kimag,kreal) = abs(Ehat);
        absE(kimag,kreal) = abs(E);
        absF(kimag,kreal) = abs(f);
    end
end

figure(2)
fs = 14;
subplot(221)
imagesc(alpha,beta,absR,[0 1]); % In the plotting we show up until maximum
grid on
colorbar
axis image
axis xy
xlabel('real','fontsize',fs);
ylabel('imag','fontsize',fs);
```

```
title('|R(z)|','fontsize',fs)

subplot(222)
imagesc(alpha,beta,absE,[0 1]); % In the plotting we show up until maximum
grid on
colorbar
axis image
axis xy
xlabel('real','fontsize',fs);
ylabel('imag','fontsize',fs);
title('|E(z)|','fontsize',fs)

subplot(223)
imagesc(alpha,beta,absEhat,[0 1]); % In the plotting we show up until maximum
grid on
colorbar
axis image
axis xy
xlabel('real','fontsize',fs);
ylabel('imag','fontsize',fs);
title('|Ehat(z)|','fontsize',fs)

subplot(224)
imagesc(alpha,beta,absEhatmE,[0 1]); % In the plotting we show up until maximum
grid on
colorbar
axis image
axis xy
xlabel('real','fontsize',fs);
ylabel('imag','fontsize',fs);
title('|Ehat(z) - E(z)|','fontsize',fs)

close all; clear all;

lw = 3;
%% The Prey

if (0)
    tspan = [0 50];
    Y0 = [2 ; 2];
    [Tout,Xout,Eout] = RK3(@DepPrey,tspan,Y0,h, mu);  % @DepPrey @VanDelPol

    figure()
    subplot(3,1,1);
    plot(Xout(:,2), Xout(:,1), 'color',rand(1,3), 'LineWidth',lw)
    subplot(3,1,2);
    plot(Tout, Xout(:,1), 'color',rand(1,3), 'LineWidth',lw)
    subplot(3,1,3);
    plot(Tout, Xout(:,2), 'color',rand(1,3), 'LineWidth',lw)
end

%% The VanderPol Comparison with ode15s


%% Normalize by y(n)

mu = 3;
h = 0.01;
tspan = [0 5*mu];
Y0 = [2 ; 0];
```

```
[Tout_RK3,Xout_RK3,Eout_RK3] = RK3(@VanDelPol,tspan,Y0,h, mu);  % @DepPrey @VanDelPol

% method = 'DOPRI54';
% solver = ERKSolverErrorEstimationParameters(method);
% [Tout_RK4,Xout_RK4,Eout_RK4] = ERKSolverErrorEstimation(@VanDelPol,tspan,Y0,h,solver,mu);

[Tout_ode15s,Xout_ode15s] = ode15s(@(t,y) VanDelPol(t,y,mu),tspan,Y0);

figure()
subplot(3,1,1);
hold on
    plot(Xout_RK3(:,2), Xout_RK3(:,1), 'color',rand(1,3), 'LineWidth',lw)
    plot(Xout_ode15s(:,2), Xout_ode15s(:,1), 'color',rand(1,3), 'LineWidth',lw)
    legend('RK3','ode15s')
    ylabel('x_2','FontSize',12,'FontWeight','bold')
    xlabel('x_1','FontSize',12,'FontWeight','bold')
hold off

subplot(3,1,2);
hold on
    plot(Tout_RK3, Xout_RK3(:,1), 'color',rand(1,3), 'LineWidth',lw)
    plot(Tout_ode15s, Xout_ode15s(:,1), 'color',rand(1,3), 'LineWidth',lw)
    legend('RK3','ode15s')
    ylabel('x_1','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')
hold off

subplot(3,1,3);
hold on
    plot(Tout_RK3, Xout_RK3(:,2), 'color',rand(1,3), 'LineWidth',lw)
    plot(Tout_ode15s, Xout_ode15s(:,2), 'color',rand(1,3), 'LineWidth',lw)
    legend('RK3','ode15s')
    ylabel('x_2','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')
hold off


%% Error figure
figure();

subplot(3,1,1); title('Estimated Local Error 1');
hold on
    plot(Tout_RK3, abs(Eout_RK3(:,1)), 'color',rand(1,3), 'LineWidth',lw)
    legend('Estimated Local Error', 'Actual Local Error', 'Normalized Local Error')
    ylabel('log(|e_1|)','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')

hold off

subplot(3,1,2); title('Estimated Local Error 2');
hold on
    plot(Tout_RK3, abs(Eout_RK3(:,2)), 'color',rand(1,3), 'LineWidth',lw)
    legend('Estimated Local Error', 'Actual Local Error', 'Normalized Local Error')
    ylabel('log(|e_2|)','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')

hold off

subplot(3,1,3); title('Estimated Local Error');
hold on
    plot(Tout_RK3, sqrt(Eout_RK3(:,2).^2 + Eout_RK3(:,1).^2 ), 'color',rand(1,3), 'LineWidth',lw)
    legend('Estimated Local Error', 'Actual Local Error', 'Normalized Local Error')
    ylabel('log(|Error|)','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')

hold off

function [Tout,Xout,Eout] = RK3(func,tspan,Y0,h, varargin)
%
% This function solves a general first-order Initial Value Problem
% of the form
%                 u? = f(u,t),  u(tstart) = eta
%
% using Euler?s Method in n steps (constant step size).
%
% INPUT:
%    func  : a function handle to function f(u,t)
%    tspan : a 1x2 array of the form [tstart tend]
%    N     : total number of steps in tspan
%    y0    : initialvalue(s)
%    param : parameters to be passed to func
%

%% Define the parameters of the model
A = [0 0 0;
    1/2 0 0;
    -1 2 0];
b = [1/6; 2/3; 1/6];
c = [0; 1; 1/2];
d = [1/12; -1/6; 1/12];
s = 3;

AT = A.';
% Parameters related to constant step size
hAT = h*AT;
hb  = h*b;
hc  = h*c;
hd  = h*d;

% Size parameters
x   = Y0;                % Initial state
t   = tspan(1);          % Initial time
tf  = tspan(end);        % Final time
N = int64((tf-t)/h);            % Number of steps
nx = length(Y0);         % System size (dim(x))

% Allocate memory
T   = zeros(1,s);        % Stage T
X   = zeros(nx,s);       % Stage X
F   = zeros(nx,s);       % Stage F

Tout = zeros(N+1,1);     % Time for output
Xout = zeros(N+1,nx);    % States for output
Eout = zeros(N+1,nx);    % Errors for output

% Algorithm starts here
Tout(1) = t;
Xout(1,:) = x';
for n=1:N
    % Stage 1
    T(1)   = t;
    X(:,1) = x;
    F(:,1) = func(T(1),X(:,1),varargin{:});

    % Stage 2,3,...,s
    T(2:s) = t + hc(2:s);
    for i=2:s
        X(:,i) = x + F(:,1:i-1)*hAT(1:i-1,i);
        F(:,i) = feval(func,T(i),X(:,i),varargin{:});
    end

    % Next step
    t = t + h;
    x = x + F*hb;
    e = F*hd;

    % Save output
    Tout(n+1) = t;
    Xout(n+1,:) = x';
    Eout(n+1,:) = e';
end

function [tnList,ynList] = RK4(func,tspan,N,Y0)
%
% This function solves a general first-order Initial Value Problem
% of the form
%                 dot(Y) = F(Y,t),  Y(tstart) = Y0
%
% using classic Runge Kutta method.
%
% INPUT:
%    func  : a function handle to function f(y,t)
%    tspan : a 1x2 array of the form [tstart tend]
%    N     : total number of steps in tspan
%    Y0   : initialvalue(s)
%
sizeY = size(Y0);
Ndim = sizeY(1);
sizeTspan = size(tspan);
Ninit = sizeTspan(2);  % We can be given only the end time, then the begining is 0

ynList = zeros(Ndim,N+1);
tnList = zeros(1,N+1);

%% Initialization
if (Ninit == 1)
    tbegin = 0;
    tend = tspan;
elseif(Ninit == 2) % [tbegin tend]
    tbegin = tspan(1,1);
    tend = tspan(1,2);
end
dt = (tend - tbegin)/N;

tnList(1) = tbegin;
ynList(:,1) = Y0;

%% Loop

for k = 1:N
    s1 = feval(func,tnList(k),ynList(:,k));  % Explicit Euleer
    s2 = feval(func,tnList(k) + dt/2 ,ynList(:,k) + (dt/2) * s1);  % Midpoint method
    s3 = feval(func,tnList(k) + dt/2 ,ynList(:,k) + (dt/2) * s2);  % Recursive of Midpoint method
    s4 = feval(func,tnList(k) + dt,ynList(:,k) + s3*dt); % Slope in the predictive point as if used 

    ynList(:,k+1) = ynList(:,k)+ (dt/6)*(s1 + 2*s2 + 2*s3 + s4);
    tnList(k+1) = tnList(k)+dt;
end

function dydt = func(t,y,params)
%Retrun the function dy/dt=f(t,y(t))
lambda = -1;
dydt=lambda*y;
end

function [ gradY ] = VanDelPol( t, Y, args )
%VANDELPOL Summary of this function goes here
%   Detailed explanation goes here
    mu = args;

    % Syntax: xdot = PreyPredator(t,x,a,b)
    gradY = zeros(2,1);
    gradY(1) = Y(2);
    gradY(2) = mu*(1-power(Y(1),2))*Y(2) - Y(1);

end

function [f,g] = VanderPolFun(t,x,mu)

f = [x(2); mu*(1-x(1)*x(1))*x(2)-x(1)];
g = x;
```

A.4 Question 4

```
%% Question 1: mu = 3 P-controller
close all; clear all;

% Paramters
tspan = [0 15];
N = 1000;
y0 = [1;1];
method = 'DOPRI54';
solver = ERKSolverErrorEstimationParameters(method);
h = (tspan(2)-tspan(1))/N;
abstol=10e-7;
reltol = 10e-7;

% Solvers
[tnList1,ynList1,hList1,rList1,nfun1] = ExplicitEulersAdaptiveStep(...
    @VanDerPol,tspan,N,y0,abstol,reltol,'P');
[tnList2,ynList2,hList2,rList2,nfun2] = ImplicitEulersAdaptiveStep(...
    @VanDerPol,@jacVanDerPol,tspan,N,y0,abstol,reltol,'P');
[tnList3,ynList3,hList3,rList3,nfun3] = RK4AdaptiveStep(...
    @VanDerPol,tspan,N,y0,abstol,reltol);
[tnList4,ynList4,hList4,rList4,nfun4] = ImplicitTrapezoidAdaptiveStep(...
    @VanDerPol,@jacVanDerPol,tspan,N,y0,abstol,reltol,'P');
% [tnList,ynList,Eout] = ERKSolverAdaptiveStep(...
%     @VanDerPol,tspan,y0,h,solver,abstol,reltol);

% Plotting
figure(1)
lw = 2;
subplot(2,1,1)
hold on
    plot(tnList1(1,:),ynList1(1,:),'-.','LineWidth',lw)
    plot(tnList2(1,:),ynList2(1,:),'-.','LineWidth',lw+1)
    plot(tnList3(1,:),ynList3(1,:),':','LineWidth',lw)
    plot(tnList4(1,:),ynList4(1,:),':','LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler',...
        'Trapezoidal','RK4');
    set(leg,'FontSize',14);
    ylabel('x','FontSize',12,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',12,'FontWeight','bold')
    grid on
hold off

subplot(2,1,2)
hold on
    plot(tnList1(1,:),ynList1(2,:),'-.','LineWidth',lw)
    plot(tnList2(1,:),ynList2(2,:),'-.','LineWidth',lw+1)
    plot(tnList3(1,:),ynList3(2,:),':','LineWidth',lw)
    plot(tnList4(1,:),ynList4(2,:),':','LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler',...
        'Trapezoidal','RK4');
    set(leg,'FontSize',14);
    ylabel('y','FontSize',12,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',12,'FontWeight','bold')
    grid on
hold off

figure(2)
lw = 2;
subplot(2,1,1)
hold on
    plot(tnList1(2:end),hList1,'-.','LineWidth',lw)
    plot(tnList2(2:end),hList2,'-.','LineWidth',lw)
    plot(tnList3(2:end),hList3,':','LineWidth',lw)
    plot(tnList4(2:end),hList4,'--','LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler',...
        'Trapezoidal','RK4');
    set(leg,'FontSize',14);
    ylabel('h','FontSize',14,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',14,'FontWeight','bold')
    grid on
hold off

subplot(2,1,2)
ve = ones(1,length(tnList1)-1);
hold on
    plot(tnList1(2:end),rList1,'-.','LineWidth',lw)
    plot(tnList2(2:end),rList2,'-.','LineWidth',lw)
    plot(tnList3(2:end),rList3,':','LineWidth',lw)
    plot(tnList4(2:end),rList4,'--','LineWidth',lw)
    plot(tnList1(2:end),ve,'LineWidth',3)
    leg = legend('Explicit Euler','Implicit Euler',...
        'Trapezoidal','RK4','MaxTolerance');
    set(leg,'FontSize',14);
    ylim([0 1.3])
    ylabel('r','FontSize',14,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',14,'FontWeight','bold')
    grid on
hold off

%% Question 2: mu = 3 PI
close all; clear all;
```

```
% Paramters
tspan = [0 15];
N = 1000;
y0 = [1;1];
method = 'DOPRI54';
solver = ERKSolverErrorEstimationParameters(method);
h = (tspan(2)-tspan(1))/N;
abstol=10e-5;
reltol = 10e-5;

% Solvers
[tnList1,ynList1,hList1,rList1,nfun1] = ExplicitEulersAdaptiveStep(...
    @VanDerPol,tspan,N,y0,abstol,reltol,'PI');
[tnList4,ynList4,hList4,rList4,nfun4] = ImplicitTrapezoidAdaptiveStep(...
    @VanDerPol,@jacVanDerPol,tspan,N,y0,abstol,reltol,'PI');

% Plotting
figure(1)
lw = 2;
subplot(2,1,1)
hold on
    plot(tnList1(1,:),ynList1(1,:),'-.','LineWidth',lw)
    plot(tnList4(1,:),ynList4(1,:),':','LineWidth',lw)
    leg = legend('Explicit Euler','Trapezoidal');
    set(leg,'FontSize',14);
    ylabel('x','FontSize',12,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',12,'FontWeight','bold')
    grid on
hold off

subplot(2,1,2)
hold on
    plot(tnList1(1,:),ynList1(2,:),'-.','LineWidth',lw)
    plot(tnList4(1,:),ynList4(2,:),':','LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler',...
        'Trapezoidal','RK4');
    set(leg,'FontSize',14);
    ylabel('y','FontSize',12,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',12,'FontWeight','bold')
    grid on
hold off

figure(2)
lw = 2;
subplot(2,1,1)
hold on
    plot(tnList1(2:end),hList1,'-.','LineWidth',lw)
    plot(tnList2(2:end),hList2,'-.','LineWidth',lw)
    plot(tnList3(2:end),hList3,':','LineWidth',lw)
    plot(tnList4(2:end),hList4,'--','LineWidth',lw)
    leg = legend('Explicit Euler','Implicit Euler',...
        'Trapezoidal','RK4');
    set(leg,'FontSize',14);
    ylabel('h','FontSize',14,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',14,'FontWeight','bold')
    grid on
hold off

subplot(2,1,2)
ve = ones(1,length(tnList1)-1);
hold on
```

```
    plot(tnList1(2:end),rList1,'-.','LineWidth',lw)
    plot(tnList2(2:end),rList2,'-.','LineWidth',lw)
    plot(tnList3(2:end),rList3,':','LineWidth',lw)
    plot(tnList4(2:end),rList4,'--','LineWidth',lw)
    plot(tnList1(2:end),ve,'LineWidth',3)
    leg = legend('Explicit Euler','Implicit Euler',...
        'Trapezoidal','RK4','MaxTolerance');
    set(leg,'FontSize',14);
    ylim([0 1.3])
    ylabel('r','FontSize',14,'FontWeight','bold')
    xlabel('time(seconds)','FontSize',14,'FontWeight','bold')
    grid on
hold off

function [yn1_n1,nfun] = newtonODEadaptive(func,Jacob,y0,tol,tn1,yn,h)
%Newton root finding algorithm
%   Fibd the roots of F(yn1) = yn1 - h*f(yn1) - yn
%   where yn1 is the unknown variable
% INPUT:
%    func  : a function handle to function f(y,t)
%    Jacob : jacobian of f(y,t)
%    y0    :  intial guess for the root i.e yn1_0
%    param : parameters to be passed to func
%    tol : tolerance for the solution
%    yn,tn1,h; are constant paramters of the fucntion F(yn1)
% OUTPUT:
%    yn1_n1  : estimated root

nfun = 0;
yn1_n = y0;
maxIt = 100;
err = 1000;
it = 0;
I = eye(length(y0));

while((it < maxIt) && (norm(err,'inf') > tol))
    it = it +1;
    fprime = I-h*feval(Jacob,tn1,yn1_n);
    f = yn1_n-h*feval(func,tn1,yn1_n)-yn;
    yn1_n1 = yn1_n - fprime\f;
    f = yn1_n1-h*feval(func,tn1,yn1_n1)-yn;
    nfun = nfun+2;
    err = f;
    yn1_n = yn1_n1;
end
end

function [y1] = RungeKuttaStep(func,t,y,h,f)
%SINGLE STEP RUNGE-KUTTA
s1 = f;  % Explicit Euleer
s2 = feval(func,t + h/2 ,y + (h/2) * s1);  % Midpoint method
s3 = feval(func,t + h/2 ,y + (h/2) * s2);  % Recursive of Midpoint method
s4 = feval(func,t + h,y + s3*h); % Slope in the predictive point as if used Euler with s3 as slope

y1 = y+ (h/6)*(s1 + 2*s2 + 2*s3 + s4);

end

function [tnList,ynList,hList,rList,nfun] = ExplicitEulersAdaptiveStep(func,...
    tspan,N,Y0,abstol,reltol,controller)
%
% This function solves a general first-order Initial Value Problem
% of the form
%                dot_y = f(y,t),  y(tstart) = tbegin
%
% using Explicit Euler in n steps (adaptive step size).
%
% INPUT:
%    func  : a function handle to function f(u,t)
%    tspan : a 1x2 array of the form [tstart tend]
%    N      : paramter for calculating first step size
%    y0    : initialvalue(s)
%    abstol : absolute tolerance
%    reltol : relative tolerance
% OUTPUT:
%    tnList  : time
%    ynList : solution
%    hList      : each step size
%    rList   : estimated error each step
%    nfun : number of function evaluations
%

sizeY = size(Y0);
Ndim = sizeY(1);
sizeTspan = size(tspan);
Ninit = sizeTspan(2);  % We can be given only the end time, then the begining is 0

ynList = [];
tnList = [];
rList = [];
hList = [];

%% Initialization
if (Ninit == 1)
    tbegin = 0;
    tend = tspan;
elseif(Ninit == 2) % [tbegin tend]
    tbegin = tspan(1,1);
    tend = tspan(1,2);
end
h = (tend - tbegin)/N;

tnList(1) = tbegin;
ynList(:,1) = Y0;
%%error estimation and control paramters
epstol = 0.8;
facmin = 0.1;
facmax = 5;

if controller == 'PI'
    a = 1/6;
    b = 1/6;
    c = 1/2;
else
    a = 1/2;
    b = 0;
    c = 0;
end

%% Loop
k=1;
nfun = 0;
while tnList(k) < tend
```

```
    %In order to compute until tend
    if (tnList(k)+h>tend)
        h = tend-tnList(k);
    end
    nfun = nfun+1;
    f = feval(func,tnList(k),ynList(:,k));

    acceptedStep = 0;
    while ~acceptedStep
        nfun = nfun+1;
        y = ynList(:,k)+h*f;

        hm = h/2;
        ym = ynList(:,k)+hm*f;
        f = feval(func,tnList(k)+hm,ym);
        y_hat = ym + hm*f;

        e = abs(y_hat - y);
        r = max(e./max(abstol,abs(y_hat).*reltol));
        if r<=1.0
            acceptedStep = 1;
            hList(k) = h;
            rList(k) = r;
            ynList(:,k+1) = y_hat;
            tnList(k+1) = tnList(k)+h;
        end
        %hnew = h*sqrt(wanted_error/current_error)
        if k == 1
            con = ((epstol/r)^a);
        else
            con = ((epstol/r)^a)*((epstol/rList(k-1))^b)*((h/hList(k-1))^-c);
        end
        h = max(facmin, min(con,facmax))*h;
    end
    k = k+1;
end

 function [tnList,ynList,hList,rList,nfun] = ImplicitEulersAdaptiveStep(func,...
     Jacob,tspan,N,Y0,abstol,reltol,controller)
%
% This function solves a general first-order Initial Value Problem
% of the form
%                 dot_y = f(y,t),  y(tstart) = tbegin
%
% using Implicit euler in n steps (adaptive step size).
%
% INPUT:
%    func   : a function handle to function f(u,t)
%    tspan : a 1x2 array of the form [tstart tend]
%    N      : paramter for calculating first step size
%    y0    : initialvalue(s)
%    abstol : absolute tolerance
%    reltol : relative tolerance
% OUTPUT:
%    tnList  : time
%    ynList : solution
%    hList      : each step size
%    rList   : estimated error each step
%    nfun : number of function evaluations
%
```

```
sizeY = size(Y0);
Ndim = sizeY(1);
sizeTspan = size(tspan);
Ninit = sizeTspan(2);

%% Initialization
if (Ninit == 1)
    tbegin = 0;
    tend = tspan;
elseif(Ninit == 2) % [tbegin tend]
    tbegin = tspan(1,1);
    tend = tspan(1,2);
end
h = (tend - tbegin)/N;

ynList = [];
tnList = [];
rList = [];
hList = [];
tnList(1) = tbegin;
ynList(:,1)=Y0;

tol = 10e-5; % Newtons method tolerance

%%error estimation and control paramters
epstol = 0.8;
facmin = 0.1;
facmax = 5;

if controller == 'PI'
    a = 1/6;
    b = 1/6;
    c = 1/2;
else
    a = 1/2;
    b = 0;
    c = 0;
end

%% Loop
k=1;
nfun = 0;
while tnList(k) < tend

    %In order to compute until tend
    if (tnList(k)+h>tend)
        h = tend-tnList(k);
    end

    f = feval(func,tnList(k),ynList(:,k));
    nfun = nfun+1;

    acceptedStep = 0;
    it = 0;
    while ~acceptedStep
        it = it+1;
        %y(n+1) in one step
        y_guess = ynList(:,k)+h*f; %Forwards euler
        [y,calls] = newtonODEadaptive(func,Jacob,y_guess,tol,tnList(k)+h,ynList(:,k),h);
        nfun = nfun+calls+1;
        %y(n+1) in two step
        hm = h/2;
        ym_guess = ynList(:,k)+hm*f;
        [ym,calls] = newtonODEadaptive(func,Jacob,ym_guess,...
            tol,tnList(k)+hm,ynList(:,k),hm);
        nfun = nfun + calls +1;
        f = feval(func,tnList(k)+hm,ym);
        y_hat_guess = ynList(:,k)+hm*f;
        [y_hat,calls] = newtonODEadaptive(func,Jacob,y_hat_guess,...
            tol,tnList(k)+h,ym,hm);
        nfun = nfun +calls+1;

        e = abs(y_hat - y);
        r = max(e./max(abstol,abs(y_hat).*reltol));
        if r<=1.0 || it > 1000
            rList(k) = r;
            hList(k) = h;
            acceptedStep = 1;
            ynList(:,k+1) = y_hat;
            tnList(k+1) = tnList(k)+h;
        end
        %hnew = h*sqrt(wanted_error/current_error)
        if k == 1
            con = ((epstol/r)^a);
        else
            con = ((epstol/r)^a)*((epstol/rList(k-1))^b)*((h/hList(k-1))^-c);
        end
        h = max(facmin, min(con,facmax))*h;
    end
    k = k+1;
end

function [tnList,ynList,hList,rList,nfun] = ImplicitTrapezoidAdaptiveStep...
    (func,Jacob,tspan,N,Y0,abstol,reltol,controller)
%
% This function solves a general first-order Initial Value Problem
% of the form
%                dot_y = f(y,t),  y(tstart) = tbegin
%
% using Trapezoid Method in n steps (constant step size).
%
% INPUT:
%    func  : a function handle to function f(u,t)
%    tspan : a 1x2 array of the form [tstart tend]
%    N     : total number of steps in tspan
%    y0   : initialvalue(s)
%    param : parameters to be passed to func
%
sizeY = size(Y0);
Ndim = sizeY(1);
sizeTspan = size(tspan);
Ninit = sizeTspan(2);  % We can be given only the end time, then the begining is 0

ynList = [];
tnList = [];
rList = [];
hList = [];

%% Initialization
if (Ninit == 1)
```

```
    tbegin = 0;
    tend = tspan;
elseif(Ninit == 2) % [tbegin tend]
    tbegin = tspan(1,1);
    tend = tspan(1,2);
end
h = (tend - tbegin)/N;

tnList(1) = tbegin;
ynList(:,1) = Y0;
tol = 10e-5;
%%error estimation and control paramters
epstol = 0.8;
facmin = 0.1;
facmax = 5;

if controller == 'PI'
    a = 1/6;
    b = 1/6;
    c = 1/2;
else
    a = 1/3;
    b = 0;
    c = 0;
end

%% Loop
k=1;
nfun = 0;
while tnList(k) < tend
    %In order to compute until tend
    if (tnList(k)+h>tend)
        h = tend-tnList(k);
    end
    nfun = nfun+1;
    f = feval(func,tnList(k),ynList(:,k));

    acceptedStep = 0;
    while ~acceptedStep
        %%%%%%%%%in one step
        yn_guess = ynList(:,k) + h*f;
        [yn,calls] = newtonODEadaptive(func,Jacob,...
            yn_guess,tol,tnList(k)+h,ynList(:,k),h);
        nfun = nfun+calls+1;
        fn = feval(func,tnList(k) + h ,yn);
        y = ynList(:,k)+ (h/2)*(fn + f);

        %%%%%%%%in two steps
        hm = h/2;
        yn_guess = ynList(:,k) + hm*f;
        [yn,calls] = newtonODEadaptive(func,Jacob,...
            yn_guess,tol,tnList(k)+hm,ynList(:,k),hm);
        fn = feval(func,tnList(k) + hm ,yn);
        nfun = nfun+calls+1;
        ym = ynList(:,k)+ (hm/2)*(fn + f);

        fn1 = feval(func,tnList(k) + hm ,ym);
        yn_guess = ym + hm*fn1;
        [yn,calls] = newtonODEadaptive(func,Jacob,...
            yn_guess,tol,tnList(k)+h,ym,hm);
        nfun = nfun+calls+2;
        fn = feval(func,tnList(k) + h ,yn);
        y_hat = ym + (hm/2)*(fn + fn1);

        e = abs(y_hat - y);
        r = max(e./max(abstol,abs(y_hat).*reltol));
        if r<=1.0
            acceptedStep = 1;
            hList(k) = h;
            rList(k) = r;
            ynList(:,k+1) = y_hat;
            tnList(k+1) = tnList(k)+h;
        end
        %hnew = h*sqrt(wanted_error/current_error)
        if k == 1
            con = ((epstol/r)^a);
        else
            con = ((epstol/r)^a)*((epstol/rList(k-1))^b)*((h/hList(k-1))^-c);
        end
        h = max(facmin, min(con,facmax))*h;
    end
    k = k+1;
end

function [Tout,Xout,Eout] = ...
        ERKSolverErrorEstimation(fun,tspan,x0,h,solver,varargin)
% ERKSOLVERERRORESTIMATION  Fixed step size ERK solver with error est.
%
%                           Solves ODE systems in the form dx/dt = f(t,x)
%                           with x(t0) = x0.
%
% Syntax:
% [Tout,Xout,Eout]=ERKSolverErrorEstimation(fun,tspan,x0,h,solver,varargin)

% Solver Parameters
s  = solver.stages;     % Number of stages in ERK method
AT = solver.AT;         % Transpose of A-matrix in Butcher tableau
b  = solver.b;          % b-vector in Butcher tableau
c  = solver.c;          % c-vector in Butcher tableau
d  = solver.d;

% Parameters related to constant step size
hAT = h*AT;
hb  = h*b;
hc  = h*c;
hd  = h*d;

% Size parameters
x  = x0;                % Initial state
t  = tspan(1);          % Initial time
tf = tspan(end);        % Final time
N = int64((tf-t)/h);           % Number of steps
nx = length(x0);        % System size (dim(x))

% Allocate memory
T  = zeros(1,s);        % Stage T
X  = zeros(nx,s);       % Stage X
F  = zeros(nx,s);       % Stage F

Tout = zeros(N+1,1);    % Time for output
Xout = zeros(N+1,nx);   % States for output
Eout = zeros(N+1,nx);   % Errors for output
```

```
% Algorithm starts here
Tout(1) = t;
Xout(1,:) = x';
for n=1:N
    % Stage 1
    T(1)   = t;
    X(:,1) = x;
    F(:,1) = fun(T(1),X(:,1),varargin{:});

    % Stage 2,3,...,s
    T(2:s) = t + hc(2:s);
    for i=2:s
        X(:,i) = x + F(:,1:i-1)*hAT(1:i-1,i);
        F(:,i) = feval(fun,T(i),X(:,i),varargin{:});
    end

    % Next step
    t = t + h;
    x = x + F*hb;
    e = F*hd;

    % Save output
    Tout(n+1) = t;
    Xout(n+1,:) = x';
    Eout(n+1,:) = e';
end

function [tnList,ynList,Eout] = ...
        ERKSolverAdaptiveStep(fun,tspan,Y0,h,solver,abstol,reltol,varargin)
% ERKSOLVERERRORESTIMATION  Fixed step size ERK solver with error est.
%
%                           Solves ODE systems in the form dx/dt = f(t,x)
%                           with x(t0) = x0.
%
% Syntax:
% [Tout,Xout,Eout]=ERKSolverErrorEstimation(fun,tspan,x0,h,solver,varargin)

% Solver Parameters
s  = solver.stages;     % Number of stages in ERK method
AT = solver.AT;         % Transpose of A-matrix in Butcher tableau
b  = solver.b;          % b-vector in Butcher tableau
c  = solver.c;          % c-vector in Butcher tableau
d  = solver.d;

% Parameters related to constant step size
hAT = h*AT;
hb  = h*b;
hc  = h*c;
hd  = h*d;

% Size parameters
x  = Y0;                % Initial state
t  = tspan(1);          % Initial time
tend = tspan(end);        % Final time
N = int64((tend-t)/h);            % Number of steps
nx = length(Y0);        % System size (dim(x))

% Allocate memory
T  = [];        % Stage T
X  = [];       % Stage X
```

```
F  = [];        % Stage F

tnList = [];     % Time for output
ynList = [];    % States for output
Eout = [];    % Errors for output

%%error estimation and control paramters
epstol = 0.8;
facmin = 0.1;
facmax = 5;
kpow = 1/6;

% Algorithm starts here
tnList(1) = t;
ynList(:,1) = x';
k = 1;
while tnList(k)<tend

    if (tnList(k)+h>tend)
            h = tend-tnList(k);
    end

    acceptedStep = 0;
    while ~acceptedStep

        hAT = h*AT;
        hb  = h*b;
        hc  = h*c;
        hd  = h*d;

        % Stage 1
        T(1)   = t;
        X(:,1) = x;
        F(:,1) = fun(T(1),X(:,1),varargin{:});

        % Stage 2,3,...,s
        T(2:s) = t + hc(2:s);
        for i=2:s
            X(:,i) = x + F(:,1:i-1)*hAT(1:i-1,i);
            F(:,i) = feval(fun,T(i),X(:,i),varargin{:});
        end

        % Next step
        t = t + h;
        x = x + F*hb;
        e = F*hd;
        r = max(e./max(abstol,abs(x).*reltol));
        if k == 2
            k
        end
        if r<=1.0
            acceptedStep = 1;
%             hList(k) = h;
%             rList(k) = r;
            ynList(:,k+1) = x;
            tnList(k+1) = tnList(k)+h;
%             Eout(n+1,:) = e';

        end
        %hnew = h*sqrt(wanted_error/current_error)
        h = max(facmin, min((epstol/r)^kpow,facmax))*h;
    end
    k = k+1;
    tnList(k)
    k
end
% while tnList(k) < tend
%
%     if (tnList(k)+h>tend)
%             h = tend-tnList(k);
%     end
%
%     acceptedStep = 0;
%     while ~acceptedStep
%         %In order to compute until tend
%
%         hAT = h*AT;
%         hb  = h*b;
%         hc  = h*c;
%         hd  = h*d;
%         % Stage 1
%         T(1)   = t;
%         X(:,1) = x;
%         F(:,1) = fun(T(1),X(:,1),varargin{:});
%         % Stage 2,3,...,s
%         T(2:s) = t + hc(2:s);
%         for i=2:s
%             X(:,i) = x + F(:,1:i-1)*hAT(1:i-1,i);
%             F(:,i) = feval(fun,T(i),X(:,i),varargin{:});
%         end% Next step
%         x_full = x + F*hb;
%
%         %%%%%%%%%%% 2 steps %%%%%%%%%%%%
%         hAT = (h/2)*AT;
%         hb  = (h/2)*b;
%         hc  = (h/2)*c;
%         hd  = (h/2)*d;
%         % Stage 1
%         T(1)   = t;
%         X(:,1) = x;
%         F(:,1) = fun(T(1),X(:,1),varargin{:});
%         % Stage 2,3,...,s
%         T(2:s) = t + hc(2:s);
%         for i=2:s
%             X(:,i) = x + F(:,1:i-1)*hAT(1:i-1,i);
%             F(:,i) = feval(fun,T(i),X(:,i),varargin{:});
%         end% Next step
%         t = t + (h/2);
%         x_half = x + F*hb;
%
%         %%%%%%%%%%%%%
%         % Stage 1
%         T(1)   = t;
%         X(:,1) = x_half;
%         F(:,1) = fun(T(1),X(:,1),varargin{:});
%         % Stage 2,3,...,s
%         T(2:s) = t + hc(2:s);
%         for i=2:s
%             X(:,i) = x_half + F(:,1:i-1)*hAT(1:i-1,i);
%             F(:,i) = feval(fun,T(i),X(:,i),varargin{:});
%         end% Next step
%          t = t + (h/2);
%          x_hat = x_half + F*hb;
%
%          e = abs(x_hat - x_full);
%          r = max(e./max(abstol,abs(x_hat).*reltol));
%          if r<=1.0
%              acceptedStep = 1;
% %              hList(k) = h;
% %              rList(k) = r;
%              ynList(:,k+1) = x_hat;
%              tnList(k+1) = tnList(k)+h;
% %              Eout(n+1,:) = e';
%          end
%          %hnew = h*sqrt(wanted_error/current_error)
%          h = max(facmin, min(sqrt(epstol/r),facmax))*h;
%
%      end
%      %%%%%%%%%%
%      k=k+1;
% end
```

A.5 Question 5

```
close all
clear all
clc

mu = 100;
t0 = 0;
tfinal = max(300);
tspan = [t0 tfinal];

method = 'ESDIRK23';
flag = 0;
isenspar = 1;
x0 = [2; 1];
h0 = 0.001;
absTol = 1e-6;
relTol = 1e-6;
fun  = 'VanderPolFun';
jac  = 'VanderPolJac';

[Tout_ESDIRK23,Xout_ESDIRK23,Gout_ESDIRK23,Eout_ESDIRK23, info,stats] = ESDIRK23(fun,jac,[t0 tfinal]

nStep = 1:1:info.nStep;


[Tout_RK3,Xout_RK3,Eout_RK3] = RK3(@VanDelPol,tspan,x0,h0, mu);  % @DepPrey @VanDelPol

% method = 'DOPRI54';
% solver = ERKSolverErrorEstimationParameters(method);
% [Tout_RK4,Xout_RK4,Eout_RK4] = ERKSolverErrorEstimation(@VanDelPol,tspan,Y0,h,solver,mu);

lw= 3
figure()
subplot(3,1,1);
hold on
    plot(Xout_RK3(:,2), Xout_RK3(:,1), 'color',rand(1,3), 'LineWidth',lw)
    plot(Xout_ESDIRK23(:,2), Xout_ESDIRK23(:,1), 'color',rand(1,3), 'LineWidth',lw)
    legend('RK3','ESDIRK23')
    ylabel('x_2','FontSize',12,'FontWeight','bold')
    xlabel('x_1','FontSize',12,'FontWeight','bold')
hold off

subplot(3,1,2);
hold on
    plot(Tout_RK3, Xout_RK3(:,1), 'color',rand(1,3), 'LineWidth',lw)
    plot(Tout_ESDIRK23, Xout_ESDIRK23(:,1), 'color',rand(1,3), 'LineWidth',lw)
    legend('RK3','ESDIRK23')
    ylabel('x_1','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')
hold off

subplot(3,1,3);
hold on
    plot(Tout_RK3, Xout_RK3(:,2), 'color',rand(1,3), 'LineWidth',lw)
    plot(Tout_ESDIRK23, Xout_ESDIRK23(:,2), 'color',rand(1,3), 'LineWidth',lw)
    legend('RK3','ESDIRK23')
    ylabel('x_2','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')
hold off

%% Individual figure

figure()
subplot(3,1,1);
hold on

    plot(Xout_ESDIRK23(:,2), Xout_ESDIRK23(:,1), 'color',rand(1,3), 'LineWidth',lw)
    legend('ESDIRK23')
    ylabel('x_2','FontSize',12,'FontWeight','bold')
    xlabel('x_1','FontSize',12,'FontWeight','bold')
hold off

subplot(3,1,2);
hold on

    plot(Tout_ESDIRK23, Xout_ESDIRK23(:,1), 'color',rand(1,3), 'LineWidth',lw)
    legend('RK3','ESDIRK23')
    ylabel('x_1','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')
hold off

subplot(3,1,3);
hold on

    plot(Tout_ESDIRK23, Xout_ESDIRK23(:,2), 'color',rand(1,3), 'LineWidth',lw)
    legend('ESDIRK23')
    ylabel('x_2','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')
hold off


%% Error figure
figure();

subplot(3,1,1); title('Estimated Local Error 1');
hold on
    plot(Tout_RK3, log(abs(Eout_RK3(:,1))), 'color',rand(1,3), 'LineWidth',lw)
    plot(Tout_RK3, log(abs(Eout_ESDIRK23(:,1))), 'color',rand(1,3), 'LineWidth',lw)

    legend('RK3','ESDIRK23')
    ylabel('log(|e_1|)','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')

hold off

subplot(3,1,2); title('Estimated Local Error 2');
hold on
    plot(Tout_RK3, log(abs(Eout_RK3(:,2))), 'color',rand(1,3), 'LineWidth',lw)
    plot(Tout_RK3, log(abs(Eout_RK3(:,2))), 'color',rand(1,3), 'LineWidth',lw)
    legend('RK3','ESDIRK23')
    ylabel('log(|e_2|)','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')

hold off

subplot(3,1,3); title('Estimated Local Error');
hold on
    plot(Tout_RK3, log(sqrt(Eout_RK3(:,2).^2 + Eout_RK3(:,1).^2 )), 'color',rand(1,3), 'LineWidth',lw
    plot(Tout_RK3, log(sqrt(Eout_ESDIRK23(:,2).^2 + Eout_ESDIRK23(:,1).^2 )), 'color',rand(1,3), 'Li

    legend('RK3','ESDIRK23')
    ylabel('log(|Error|)','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')

hold off


ESDIRKperformance(info, stats);

%% Explicit eulers
close all; clear all;

lw = 3;

limits = 10;
alpha = -limits:0.1:limits;
beta = -limits:0.1:limits;

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;

A = AT.';
AT = A.';

nreal = length(alpha);
nimag = length(beta);
I = eye(size(A));
e = ones(size(A,1),1);

for kreal = 1:nreal
    for kimag = 1:nimag
```

```
        z = alpha(kreal) + 1i*beta(kimag);
        tmp = (I-z*A)\e;
        R = 1 + z*b.'*tmp;
        Ehat = z*d.'*tmp;
        f = exp(z); % This is the real variation
        E = R-f;    % This is the estimated variation
        EhatmE = Ehat-E;

        absR(kimag,kreal) = abs(R);
        absEhatmE(kimag,kreal) = abs(EhatmE);
        absEhat(kimag,kreal) = abs(Ehat);
        absE(kimag,kreal) = abs(E);
        absF(kimag,kreal) = abs(f);
    end
end


figure(2)
fs = 14;
subplot(221)
imagesc(alpha,beta,absR,[0 1]); % In the plotting we show up until maximum
grid on
colorbar
axis image
axis xy
xlabel('real','fontsize',fs);
ylabel('imag','fontsize',fs);
title('|R(z)|','fontsize',fs)

subplot(222)
imagesc(alpha,beta,absE,[0 1]); % In the plotting we show up until maximum
grid on
colorbar
axis image
axis xy
xlabel('real','fontsize',fs);
ylabel('imag','fontsize',fs);
title('|E(z)|','fontsize',fs)

subplot(223)
imagesc(alpha,beta,absEhat,[0 1]); % In the plotting we show up until maximum
grid on
colorbar
axis image
axis xy
xlabel('real','fontsize',fs);
ylabel('imag','fontsize',fs);
title('|Ehat(z)|','fontsize',fs)

subplot(224)
imagesc(alpha,beta,absEhatmE,[0 1]); % In the plotting we show up until maximum
grid on
colorbar
axis image
axis xy
xlabel('real','fontsize',fs);
ylabel('imag','fontsize',fs);
title('|Ehat(z) - E(z)|','fontsize',fs)

close all
clear all
```

```
clc

mu = 100;
t0 = 0;
tfinal = max(10, 3*mu);
tspan = [t0 tfinal];

method = 'ESDIRK23';
flag = 0;
isenspar = 1;
x0 = [2; 1];
h0 = 0.001;
absTol = 1e-6;
relTol = 1e-6;
fun  = 'VanderPolFun';
jac  = 'VanderPolJac';


[Tout_ESDIRK23,Xout_ESDIRK23,Gout_ESDIRK23,Eout_ESDIRK23,info,stats] = ESDIRK(fun,jac,t0,tfinal,x0,h
[Tout_RK3,Xout_RK3,Eout_RK3] = RK3(@VanDelPol,tspan,x0,h0, mu);  % @DepPrey @VanDelPol

% method = 'DOPRI54';
% solver = ERKSolverErrorEstimationParameters(method);
% [Tout_RK4,Xout_RK4,Eout_RK4] = ERKSolverErrorEstimation(@VanDelPol,tspan,Y0,h,solver,mu);

lw= 3
figure()
subplot(3,1,1);
hold on
    plot(Xout_RK3(:,2), Xout_RK3(:,1), 'color',rand(1,3), 'LineWidth',lw)
    plot(Xout_ESDIRK23(:,2), Xout_ESDIRK23(:,1), 'color',rand(1,3), 'LineWidth',lw)
    legend('RK3','ESDIRK23')
    ylabel('x_2','FontSize',12,'FontWeight','bold')
    xlabel('x_1','FontSize',12,'FontWeight','bold')
hold off

subplot(3,1,2);
hold on
    plot(Tout_RK3, Xout_RK3(:,1), 'color',rand(1,3), 'LineWidth',lw)
    plot(Tout_ESDIRK23, Xout_ESDIRK23(:,1), 'color',rand(1,3), 'LineWidth',lw)
    legend('RK3','ESDIRK23')
    ylabel('x_1','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')
hold off

subplot(3,1,3);
hold on
    plot(Tout_RK3, Xout_RK3(:,2), 'color',rand(1,3), 'LineWidth',lw)
    plot(Tout_ESDIRK23, Xout_ESDIRK23(:,2), 'color',rand(1,3), 'LineWidth',lw)
    legend('RK3','ESDIRK23')
    ylabel('x_2','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')
hold off

%% Individual figure

figure()
subplot(3,1,1);
hold on

    plot(Xout_ESDIRK23(:,2), Xout_ESDIRK23(:,1), 'color',rand(1,3), 'LineWidth',lw)
    legend('ESDIRK23')
    ylabel('x_2','FontSize',12,'FontWeight','bold')
    xlabel('x_1','FontSize',12,'FontWeight','bold')
hold off

subplot(3,1,2);
hold on

    plot(Tout_ESDIRK23, Xout_ESDIRK23(:,1), 'color',rand(1,3), 'LineWidth',lw)
    legend('RK3','ESDIRK23')
    ylabel('x_1','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')
hold off

subplot(3,1,3);
hold on

    plot(Tout_ESDIRK23, Xout_ESDIRK23(:,2), 'color',rand(1,3), 'LineWidth',lw)
    legend('ESDIRK23')
    ylabel('x_2','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')
hold off


%% Error figure
figure();

subplot(3,1,1); title('Estimated Local Error 1');
hold on
    plot(Tout_RK3, log(abs(Eout_RK3(:,1))), 'color',rand(1,3), 'LineWidth',lw)
    plot(Tout_ESDIRK23, log(abs(Eout_ESDIRK23(:,1))), 'color',rand(1,3), 'LineWidth',lw)

    legend('RK3','ESDIRK23')
    ylabel('log(|e_1|)','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')

hold off

subplot(3,1,2); title('Estimated Local Error 2');
hold on
    plot(Tout_RK3, log(abs(Eout_RK3(:,2))), 'color',rand(1,3), 'LineWidth',lw)
    plot(Tout_ESDIRK23, log(abs(Eout_ESDIRK23(:,2))), 'color',rand(1,3), 'LineWidth',lw)
    legend('RK3','ESDIRK23')
    ylabel('log(|e_2|)','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')

hold off

subplot(3,1,3); title('Estimated Local Error');
hold on
    plot(Tout_RK3, log(sqrt(Eout_RK3(:,2).^2 + Eout_RK3(:,1).^2 )), 'color',rand(1,3), 'LineWidth',l
    plot(Tout_ESDIRK23, log(sqrt(Eout_ESDIRK23(:,2).^2 + Eout_ESDIRK23(:,1).^2 )), 'color',rand(1,3)

    legend('RK3','ESDIRK23')
    ylabel('log(|Error|)','FontSize',12,'FontWeight','bold')
    xlabel('time','FontSize',12,'FontWeight','bold')

hold off


ESDIRKperformance(info, stats);
```

```
function [Tout,Xout,Gout,Eout, info,stats] = ESDIRK(fun,jac,t0,tf,x0,h0,absTol,relTol,Method,varargi

%% ESDIRK23 Parameters
%=============================================================================
% Runge-Kutta method parameters
switch Method
    case 'ESDIRK12'
        gamma = 1;
        AT = [0 0;0 gamma];
        c  = [0; 1];
        b  = AT(:,2);
        bhat = [1/2; 1/2];
        d  = b-bhat;
        p  = 1;
        phat = 2;
        s = 2;
    case 'ESDIRK23'
        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;
    case 'ESDIRK34'
        gamma = 0.43586652150845899942;
        a31 = 0.14073777472470619619;
        a32 = -0.1083655513813208000;
        AT  = [0 gamma a31   0.10239940061991099768;
            0 gamma a32   -0.3768784522555561061;
            0 0     gamma 0.83861253012718610911;
            0 0     0     gamma                   ];
        c  = [0; 0.87173304301691799883; 0.46823874485184439565; 1];
        b  = AT(:,4);
        bhat = [0.15702489786032493710;
            0.11733044137043884870;
            0.61667803039212146434;
            0.10896663037711474985];
        d = b-bhat;
        p = 3;
        phat = 4;
        s = 4;
end


% 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(...
            'Method',    Method,  ... % carsten
            'nStage',    s,       ... % carsten
            'absTol',    'dummy',  ... % carsten
            'relTol',    'dummy',  ... % 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;

[F(:,1),g]  = feval(fun,t,x,varargin{:});
info.nFun = info.nFun+1;
[dfdx,dgdx] = feval(jac,t,x,varargin{:});
info.nJac = info.nJac+1;
FreshJacobian = true;
if (t+h)>tf
    h = tf-t;
end
hgamma = h*gamma;
dRdx = dgdx - hgamma*dfdx;
[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);
Xout = zeros(chunk,nx);
Gout = zeros(chunk,nx);
Eout = zeros(chunk,nx);

Tout(1,1) = t;
Xout(1,:) = x.';
Gout(1,:) = g.';
Eout(1,:) = zeros(1,nx);
```

```
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 = g + F(:,1:i-1)*(h*AT(1:i-1,i));

        % Initial guess for the state
         if i==2
             dt = c(i)*h;
             G = g + dt*F(:,1);
             X = x + dgdx\(G-g);
         else
             dt = c(i)*h;
             G  = g + dt*F(:,1);
             X  = x + dgdx\(G-g);
         end
        T = t+dt;

        [F(:,i),G] = feval(fun,T,X,varargin{:});
        info.nFun = info.nFun+1;
        R = G - hgamma*F(:,i) - phi;
%        rNewton = norm(R./(absTol + abs(G).*relTol),2)/sqrt(nx);
        rNewton = norm(R./(absTol + abs(G).*relTol),inf);
        Converged = (rNewton < tau);
        %iter(i) = 0; % original, if uncomment then comment line 154: iter(:) = 0;
        % Newton Iterations
        while ~Converged && ~diverging && ~SlowConvergence%iter(i)<itermax
            iter(i) = iter(i)+1;
            dX = U\(L\(R(pivot,1)));
            info.nBack = info.nBack+1;
            X = X - dX;
            rNewtonOld = rNewton;
            [F(:,i),G] = feval(fun,T,X,varargin{:});
            info.nFun = info.nFun+1;
            R = G - hgamma*F(:,i) - phi;
%            rNewton = norm(R./(absTol + abs(G).*relTol),2)/sqrt(nx);
            rNewton = norm(R./(absTol + abs(G).*relTol),inf);
            alpha = max(alpha,rNewton/rNewtonOld);
            Converged = (rNewton < tau);
            diverging = (alpha >= 1);
            SlowConvergence = (iter(i) >= itermax); % carsten
            %SlowConvergence = (alpha >= 0.5); % carsten
            %if (iter(i) >= itermax), i, iter(i), Converged, diverging, pause, end % carsten
        end
        %diverging = (alpha >= 1); % original, if uncomment then comment line 142: diverging = (alph
        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 sl
    iter(:) = 0; % carsten
    %======================================================================
    % Error and Convergence Controller
    if Converged
        % Error estimation
        e = F*(h*d);
%         r = norm(e./(absTol + abs(G).*relTol),2)/sqrt(nx);
        r = norm(e./(absTol + abs(G).*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;
            g = G;
            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,dgdx] = feval(jac,t,x,varargin{:});
            info.nJac = info.nJac+1;
            FreshJacobian = true;
            hgamma = h*gamma;
            dRdx = dgdx - hgamma*dfdx;
            [L,U,pivot] = lu(dRdx,'vector');
            info.nLU = info.nLU+1;
            hLU = h;
        elseif (abs(h-hLU)/hLU) > alphaLU
            hgamma = h*gamma;
            dRdx = dgdx-hgamma*dfdx;
            [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,dgdx] = feval(jac,t,x,varargin{:});
            info.nJac = info.nJac+1;
            FreshJacobian = true;
        end
        hgamma = h*gamma;
        dRdx = dgdx - hgamma*dfdx;
        [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)];
            Gout = [Gout; zeros(chunk,nx)];
            Eout = [Eout; zeros(chunk,nx)];
        end
        Tout(nAccept,1) = t;
        Xout(nAccept,:) = x.';
        Gout(nAccept,:) = g.';
        Eout(nAccept,:) = e;
    end
end
info.nSlowConv = length(find(stats.SlowConv)); % carsten
nAccept = info.nAccept;
Tout = Tout(1:nAccept,1);
Xout = Xout(1:nAccept,:);
Gout = Gout(1:nAccept,:);
Eout = Eout(1:nAccept,:);

function [Tout,Xout,Gout,Eout,info,stats] = ESDIRK23(fun,jac,tspan,x0,h0,absTol,relTol,Method,vararg

%% ESDIRK23 Parameters
%=========================================================================
% Runge-Kutta method parameters
Gout = 0;

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;

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

t0 = tspan(1);
tf = tspan(2);
info = struct(...
            'Method',    Method,  ... % carsten
            'nStage',    s,       ... % carsten
            'absTol',    'dummy',  ... % carsten
            'relTol',    'dummy',  ... % 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;
I = eye(nx);

F(: ,1) = feval (fun ,t,x, varargin {:}) ;

 info . nFun = info . nFun +1;
 dfdx = feval (jac ,t,x, varargin {:}) ;
 info . nJac = info . nJac +1;
 if (t+h)>tf
 h = tf -t;
 end
 hgamma = h* gamma ;
 dRdx = I - hgamma * dfdx ;
 [L,U, pivot ] = lu(dRdx ,'vector');
 info . nLU = info . nLU +1;

 iter = zeros (1,s);


% Output. Initial size of the output, it will increase if more step than
% chunk are needed.
chunk = 100;
Tout = zeros(chunk,1);
Xout = zeros(chunk,nx);
Eout = zeros(chunk,nx);

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

% While we have not reached the end of the tspan
while t<tf


    info.nStep = info.nStep+1;
    %======================================================================
    % A step in the ESDIRK method
    i=1;   % Variables with the step number in each iteration of the algo.
    diverging = false;
    SlowConvergence = false; % carsten
    alpha = 0.0;
    Converged = true;

    % While we have not reached the last step of the method and the
    % Newton's method has converged.

    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));

        % Initial guess for the state

        dt = c(i)*h;
        X = x + dt*F(: ,1);
        T = t+dt;

        F(:,i) = feval (fun ,T,X, varargin {:}) ;
        info.nFun = info.nFun+1;
        R = X - hgamma*F(:,i) - phi;
%         rNewton = norm(R./(absTol + abs(G).*relTol),2)/sqrt(nx);
        rNewton = norm(R./(absTol + abs(X).*relTol),inf);
        Converged = (rNewton < tau);
        %iter(i) = 0; % original, if uncomment then comment line 154: iter(:) = 0;


        % Newton Iterations !!!
        % Apply Newton's method iterations until it converges, for every
        % implicit step of the ESDIRK model
        while ~Converged && ~diverging && ~SlowConvergence%iter(i)<itermax
            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{:});
            info.nFun = info.nFun+1;
            R = X - hgamma*F(:,i) - phi;
%             rNewton = norm(R./(absTol + abs(G).*relTol),2)/sqrt(nx);
            rNewton = norm(R./(absTol + abs(X).*relTol),inf);
            alpha = max(alpha,rNewton/rNewtonOld);
            Converged = (rNewton < tau);
            diverging = (alpha >= 1);
            SlowConvergence = (iter(i) >= itermax); % carsten
            %SlowConvergence = (alpha >= 0.5); % carsten
            %if (iter(i) >= itermax), i, iter(i), Converged, diverging, pause, end % carsten
        end
        % diverging will have the first diverging state !!
        % It it diverges, it will get out of the previous loop.
        %diverging = (alpha >= 1); % original, if uncomment then comment line 142: diverging = (alph
        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 sl
    iter(:) = 0; % carsten


    %=====================================================================
    % Error and Convergence Controller

    % If all the implicit staged converged !!
    if Converged
        % Error estimation
        e = F*(h*d);

%         r = norm(e./(absTol + abs(G).*relTol),2)/sqrt(nx);
        r = norm(e./(absTol + abs(X).*relTol),inf);
        r = max(r,eps);
        stats.r(nstep) = r;

        % Next Step
        t = T;
        x = X;
        F(:,1) = F(:,s);

        % Just taking care that we do not go further than tspan
        h = max(1e-8,h);
        if (t+h) > tf
            h = tf-t;
        end
         % Jacobian Update Strategy
         dfdx = feval (jac ,t,x, varargin {:}) ;
         info . nJac = info . nJac +1;
         hgamma = h* gamma ;
         dRdx = I - hgamma * dfdx ;
         [L,U, pivot ] = lu(dRdx ,'vector');
         info . nLU = info . nLU +1;

    % If we were not able to converge in one of the steps...
    else % not converged
        info.nFail=info.nFail+1;
        CurrentStepAccept = false;

        hgamma = h*gamma;
        dRdx = dgdx - hgamma*dfdx;
        [L,U,pivot] = lu(dRdx,'vector');
        info.nLU = info.nLU+1;
        hLU = h;
    end

    %======================================================================
    % Storage of variables for output
       info . nAccept = info . nAccept + 1;
       nAccept = info.nAccept;
       if nAccept > length(Tout);
           Tout = [Tout; zeros(chunk,1)];
           Xout = [Xout; zeros(chunk,nx)];
           Eout = [Eout; zeros(chunk,nx)];

       end
       Tout(nAccept,1) = t;
       Xout(nAccept,:) = x.';
       Eout(nAccept,:) = e;

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

```
function solver = ERKSolverErrorEstimationParameters(method)
% ERKSOLVERERRORESTIMATIONPARAMETERS Parameter for specific ERK method
%
% Syntax: solver = ERKSolverErrorEstimationParameters(method)

switch upper(method)
    case 'RKF12'
        A = [0 0;0 0];
        b = [1; 0];
        c = [0; 1];
        d = [0.5; -0.5];
        s = 2;
    case 'RKF23'
        A = [0 0 0;1 0 0;1/4 1/4 0];
        b = [1/2; 1/2; 0];
        c = [0; 1; 1/2];
        d = [1/3; 1/3; -2/3];
        s = 3;
    case 'RKF23B'
        A = [0 0 0 0;1/4 0 0 0;-189/800 729/800 0 0;214/891 1/33 650/891 0];
        b = [41/162; 0; 800/1053; -1/78];
        c = [0; 1/4; 27/40; 1];
        d = [-23/1782; 1/33; -350/11583; 1/78];
        s = 4;
    case 'RKF45'
        s = 6;

        A = zeros(s,s);
        A(2,1) = 1/4;
        A(3,1) = 3/32;
        A(4,1) = 1932/2197;
        A(5,1) = 439/216;
        A(6,1) = -8/27;
        A(3,2) = 9/32;
        A(4,2) = -7200/2197;
        A(5,2) = -8;
        A(6,2) = 2;
        A(4,3) = 7296/2197;
        A(5,3) = 3680/513;
        A(6,3) = -3544/2565;
        A(5,4) = -845/4104;
        A(6,4) = 1859/4104;
        A(6,5) = -11/40;

        b = [25/216; 0; 1408/2565; 2197/4104; -1/5; 0];
        c = [0; 1/4; 3/8; 12/13; 1; 1/2];
        d = [-1/360; 0; 128/4275; 2197/75240; -1/50; -2/55];
    case 'RKF56'
        s = 8;

        A = zeros(s,s);
        A(2,1) = 1/6;
        A(3,1) = 4/75;
        A(4,1) = 5/6;
        A(5,1) = -8/5;
        A(6,1) = 361/320;
        A(7,1) = -11/640;
        A(8,1) = 93/640;
        A(3,2) = 16/75;
        A(4,2) = -8/3;
        A(5,2) = 144/25;
        A(6,2) = -18/5;
        A(8,2) = -18/5;
        A(4,3) = 5/2;
        A(5,3) = -4;
        A(6,3) = 407/128;
        A(7,3) = 11/256;
        A(8,3) = 803/256;
        A(5,4) = 16/25;
        A(6,4) = -11/80;
        A(7,4) = -11/160;
        A(8,4) = -11/160;
        A(6,5) = 55/128;
        A(7,5) = 11/256;
        A(8,5) = 99/256;
        A(8,7) = 1.0;

        b = [31/384; 0; 1125/2816; 9/32; 125/768; 5/66; 0; 0];
        c = [0; 1/6; 4/15; 2/3; 4/5; 1; 0; 1];
        d = [5/66; 0; 0; 0; 0; 5/66; -5/66; -5/66];
    case 'RKF78'
        s = 13;

        A = zeros(s,s);
        A(2,1) = 2/27;
        A(3,1) = 1/36;
        A(4,1) = 1/24;
        A(5,1) = 5/12;
        A(6,1) = 1/20;
        A(7,1) = -25/108;
        A(8,1) = 31/300;
        A(9,1) = 2;
        A(10,1) = -91/108;
        A(11,1) = 2383/4100;
        A(12,1) = 3/205;
        A(13,1) = -1777/4100;
        A(3,2) = 1/12;
        A(4,3) = 1/8;
        A(5,3) = -25/16;
        A(5,4) = 25/16;
        A(6,4) = 1/4;
        A(7,4) = 125/108;
        A(9,4) = -53/6;
        A(10,4) = 23/108;
        A(11,4) = -341/164;
        A(13,4) = -341/164;
        A(6,5) = 1/5;
        A(7,5) = -65/27;
        A(8,5) = 61/225;
        A(9,5) = 704/45;
        A(10,5) = -976/135;
        A(11,5) = 4496/1025;
        A(13,5) = 4496/1025;
        A(7,6) = 125/54;
        A(8,6) = -2/9;
        A(9,6) = -107/9;
        A(10,6) = 311/54;
        A(11,6) = -301/82;
        A(12,6) = -6/41;
        A(13,6) = -289/82;
        A(8,7) = 13/900;
        A(9,7) = 67/90;
```

```
        A(10,7) = -19/60;
        A(11,7) = 2133/4100;
        A(12,7) = -3/205;
        A(13,7) = 2193/4100;
        A(9,8) = 3;
        A(10,8) = 17/6;
        A(11,8) = 45/82;
        A(12,8) = -3/41;
        A(13,8) = 51/82;
        A(10,9) = -1/12;
        A(11,9) = 45/164;
        A(12,9) = 3/41;
        A(13,9) = 33/164;
        A(11,10) = 18/41;
        A(12,10) = 6/41;
        A(13,10) = 12/41;
        A(13,12) = 1;

        b = [41/840; 0; 0; 0; 0; 34/105; 9/35; 9/35; 9/280; 9/280; 41/840; 0; 0];
        c = [0; 2/27; 1/9; 1/6; 5/12; 1/2; 5/6; 1/6; 2/3; 1/3; 1; 0; 1];
        d = [41/840; 0; 0; 0; 0; 0; 0; 0; 0; 0; 41/840; -41/840; -41/840];

    case 'DOPRI54'
        s = 7;

        A = zeros(s,s);
        A(2,1) = 1/5;
        A(3,1) = 3/40;
        A(4,1) = 44/45;
        A(5,1) = 19372/6561;
        A(6,1) = 9017/3168;
        A(7,1) = 35/384;
        A(3,2) = 9/40;
        A(4,2) = -56/15;
        A(5,2) = -25360/2187;
        A(6,2) = -355/33;
        A(4,3) = 32/9;
        A(5,3) = 64448/6561;
        A(6,3) = 46732/5247;
        A(7,3) = 500/1113;
        A(5,4) = -212/729;
        A(6,4) = 49/176;
        A(7,4) = 125/192;
        A(6,5) = -5103/18656;
        A(7,5) = -2187/6784;
        A(7,6) = 11/84;

        %b = [5179/57600; 0; 7571/16695; 393/640; -92097/339200; 187/2100; 1/40];
        b = [35/384; 0; 500/1113; 125/192; -2187/6784; 11/84; 0];
        c = [0; 1/5; 3/10; 4/5; 8/9; 1; 1];
        d = [71/57600; 0; -71/16695; 71/1920; -17253/339200; 22/525; -1/40];

% % Explicit 5th order Runge-Kutta Butcher Table:
% alpha = [           0,            0,           0,          0,             0,      0 ;
%                   1/5,            0,           0,          0,             0,      0 ;
%                  3/40,         9/40,           0,          0,             0,      0 ;
%                 44/45,       -56/15,        32/9,          0,             0,      0 ;
%            19372/6561, -25360/2187, 64448/6561, -212/729,             0,      0 ;
%             9017/3168,      -355/33, 46732/5247,    49/176, -5103/18656,      0 ;
%                35/384,            0,   500/1113,   125/192,  -2187/6784, 11/84 ];
% % 4th order beta vector:
```

```
% beta4 = [ 5179/57600, 0.0, 7571/16695, 393/640, -92097/339200, 187/2100, 1/40 ];
% % 5th order beta vector:
% beta5 = [     35/384, 0.0,    500/1113, 125/192,    -2187/6784,     11/84,  0.0 ];
% % 5th order c vector:
% c = sum(alpha,2); % Sum of all rows of alpha
% % "Margin Factio" (usually 0.8-0.9)
% m = 0.8;

    case 'DOPRI87'
          c = [0 1/18 1/12 1/8 5/16 3/8 59/400 93/200 ...
                5490023248/9719169821 13/20 1201146811/1299019798 1 1]';
        A = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
         1/18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
         1/48, 1/16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
         1/32, 0, 3/32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
         5/16, 0, -75/64, 75/64, 0, 0, 0, 0, 0, 0, 0, 0, 0;
         3/80, 0, 0, 3/16, 3/20, 0, 0, 0, 0, 0, 0, 0, 0;
         29443841/614563906, 0, 0, 77736538/692538347, ...
         -28693883/1125000000, 23124283/1800000000, 0, 0, 0, 0, 0, 0, 0;
         16016141/946692911, 0, 0, 61564180/158732637, 22789713/633445777, ...
         545815736/2771057229,  -180193667/1043307555, 0, 0, 0, 0, 0, 0 ;
         39632708/573591083, 0, 0, -433636366/683701615, ...
         -421739975/2616292301, 100302831/723423059, 790204164/839813087, ...
         800635310/3783071287, 0, 0, 0, 0, 0;
         246121993/1340847787, 0 0 -37695042795/15268766246, ...
         -309121744/1061227803, -12992083/490766935, 6005943493/2108947869, ...
         393006217/1396673457, 123872331/1001029789, 0, 0, 0, 0 ;
         -1028468189/846180014, 0, 0, 8478235783/508512852, ...
         1311729495/1432422823, -10304129995/1701304382, ...
         -48777925059/3047939560, 15336726248/1032824649, ...
         -45442868181/3398467696, 3065993473/597172653, 0, 0, 0;
         185892177/718116043, 0, 0, -3185094517/667107341, ...
         -477755414/1098053517, -703635378/230739211, ...
         5731566787/1027545527, 5232866602/850066563, ...
         -4093664535/808688257, 3962137247/1805957418, ...
         65686358/487910083, 0, 0;
         403863854/491063109, 0, 0, -5068492393/434740067, ...
         -411421997/543043805, 652783627/914296604, 11173962825/925320556, ...
         -13158990841/6184727034, 3936647629/1978049680, ...
         -160528059/685178525, 248638103/1413531060, 0, 0];

    b = [14005451/335480064 0 0 0 0 -59238493/1068277825 ...
        181606767/758867731 561292985/797845732 -1041891430/1371343529 ...
        760417239/1151165299 118820643/751138087 -528747749/2220607170 1/4]';

    bhat = [13451932/455176623 0 0 0 0 -808719846/976000145 ...
            1757004468/5645159321 656045339/265891186 ...
            -3867574721/1518517206 465885868/322736535 ...
            53011238/667516719 2/45 0]';

    d = b-bhat;
    s = 13;
    otherwise
       error('Method not defined')
end

solver.AT = A';
solver.b  = b;
solver.c  = c;
solver.d  = d;
solver.stages = s;
```

```
function solver = ERKSolverErrorEstimationParameters(method)
% ERKSOLVERERRORESTIMATIONPARAMETERS Parameter for specific ERK method
%
% Syntax: solver = ERKSolverErrorEstimationParameters(method)

switch upper(method)
    case 'RKF12'
        A = [0 0;0 0];
        b = [1; 0];
        c = [0; 1];
        d = [0.5; -0.5];
        s = 2;
    case 'RKF23'
        A = [0 0 0;1 0 0;1/4 1/4 0];
        b = [1/2; 1/2; 0];
        c = [0; 1; 1/2];
        d = [1/3; 1/3; -2/3];
        s = 3;
    case 'RKF23B'
        A = [0 0 0 0;1/4 0 0 0;-189/800 729/800 0 0;214/891 1/33 650/891 0];
        b = [41/162; 0; 800/1053; -1/78];
        c = [0; 1/4; 27/40; 1];
        d = [-23/1782; 1/33; -350/11583; 1/78];
        s = 4;
    case 'RKF45'
        s = 6;

        A = zeros(s,s);
        A(2,1) = 1/4;
        A(3,1) = 3/32;
        A(4,1) = 1932/2197;
        A(5,1) = 439/216;
        A(6,1) = -8/27;
        A(3,2) = 9/32;
        A(4,2) = -7200/2197;
        A(5,2) = -8;
        A(6,2) = 2;
        A(4,3) = 7296/2197;
        A(5,3) = 3680/513;
        A(6,3) = -3544/2565;
        A(5,4) = -845/4104;
        A(6,4) = 1859/4104;
        A(6,5) = -11/40;

        b = [25/216; 0; 1408/2565; 2197/4104; -1/5; 0];
        c = [0; 1/4; 3/8; 12/13; 1; 1/2];
        d = [-1/360; 0; 128/4275; 2197/75240; -1/50; -2/55];
    case 'RKF56'
        s = 8;

        A = zeros(s,s);
        A(2,1) = 1/6;
        A(3,1) = 4/75;
        A(4,1) = 5/6;
        A(5,1) = -8/5;
        A(6,1) = 361/320;
        A(7,1) = -11/640;
        A(8,1) = 93/640;
        A(3,2) = 16/75;
        A(4,2) = -8/3;
        A(5,2) = 144/25;
        A(6,2) = -18/5;
        A(8,2) = -18/5;
        A(4,3) = 5/2;
        A(5,3) = -4;
        A(6,3) = 407/128;
        A(7,3) = 11/256;
        A(8,3) = 803/256;
        A(5,4) = 16/25;
        A(6,4) = -11/80;
        A(7,4) = -11/160;
        A(8,4) = -11/160;
        A(6,5) = 55/128;
        A(7,5) = 11/256;
        A(8,5) = 99/256;
        A(8,7) = 1.0;

        b = [31/384; 0; 1125/2816; 9/32; 125/768; 5/66; 0; 0];
        c = [0; 1/6; 4/15; 2/3; 4/5; 1; 0; 1];
        d = [5/66; 0; 0; 0; 0; 5/66; -5/66; -5/66];
    case 'RKF78'
        s = 13;

        A = zeros(s,s);
        A(2,1) = 2/27;
        A(3,1) = 1/36;
        A(4,1) = 1/24;
        A(5,1) = 5/12;
        A(6,1) = 1/20;
        A(7,1) = -25/108;
        A(8,1) = 31/300;
        A(9,1) = 2;
        A(10,1) = -91/108;
        A(11,1) = 2383/4100;
        A(12,1) = 3/205;
        A(13,1) = -1777/4100;
        A(3,2) = 1/12;
        A(4,3) = 1/8;
        A(5,3) = -25/16;
        A(5,4) = 25/16;
        A(6,4) = 1/4;
        A(7,4) = 125/108;
        A(9,4) = -53/6;
        A(10,4) = 23/108;
        A(11,4) = -341/164;
        A(13,4) = -341/164;
        A(6,5) = 1/5;
        A(7,5) = -65/27;
        A(8,5) = 61/225;
        A(9,5) = 704/45;
        A(10,5) = -976/135;
        A(11,5) = 4496/1025;
        A(13,5) = 4496/1025;
        A(7,6) = 125/54;
        A(8,6) = -2/9;
        A(9,6) = -107/9;
        A(10,6) = 311/54;
        A(11,6) = -301/82;
        A(12,6) = -6/41;
        A(13,6) = -289/82;
        A(8,7) = 13/900;
        A(9,7) = 67/90;
        A(10,7) = -19/60;
        A(11,7) = 2133/4100;
        A(12,7) = -3/205;
        A(13,7) = 2193/4100;
        A(9,8) = 3;
        A(10,8) = 17/6;
        A(11,8) = 45/82;
        A(12,8) = -3/41;
        A(13,8) = 51/82;
        A(10,9) = -1/12;
        A(11,9) = 45/164;
        A(12,9) = 3/41;
        A(13,9) = 33/164;
        A(11,10) = 18/41;
        A(12,10) = 6/41;
        A(13,10) = 12/41;
        A(13,12) = 1;

        b = [41/840; 0; 0; 0; 0; 34/105; 9/35; 9/35; 9/280; 9/280; 41/840; 0; 0];
        c = [0; 2/27; 1/9; 1/6; 5/12; 1/2; 5/6; 1/6; 2/3; 1/3; 1; 0; 1];
        d = [41/840; 0; 0; 0; 0; 0; 0; 0; 0; 0; 41/840; -41/840; -41/840];

    case 'DOPRI54'
        s = 7;

        A = zeros(s,s);
        A(2,1) = 1/5;
        A(3,1) = 3/40;
        A(4,1) = 44/45;
        A(5,1) = 19372/6561;
        A(6,1) = 9017/3168;
        A(7,1) = 35/384;
        A(3,2) = 9/40;
        A(4,2) = -56/15;
        A(5,2) = -25360/2187;
        A(6,2) = -355/33;
        A(4,3) = 32/9;
        A(5,3) = 64448/6561;
        A(6,3) = 46732/5247;
        A(7,3) = 500/1113;
        A(5,4) = -212/729;
        A(6,4) = 49/176;
        A(7,4) = 125/192;
        A(6,5) = -5103/18656;
        A(7,5) = -2187/6784;
        A(7,6) = 11/84;

        %b = [5179/57600; 0; 7571/16695; 393/640; -92097/339200; 187/2100; 1/40];
        b = [35/384; 0; 500/1113; 125/192; -2187/6784; 11/84; 0];
        c = [0; 1/5; 3/10; 4/5; 8/9; 1; 1];
        d = [71/57600; 0; -71/16695; 71/1920; -17253/339200; 22/525; -1/40];

% % Explicit 5th order Runge-Kutta Butcher Table:
% alpha = [          0,            0,           0,          0,            0,      0 ;
%                  1/5,            0,           0,          0,            0,      0 ;
%                 3/40,         9/40,           0,          0,            0,      0 ;
%                44/45,       -56/15,        32/9,          0,            0,      0 ;
%           19372/6561, -25360/2187, 64448/6561, -212/729,              0,      0 ;
%            9017/3168,      -355/33, 46732/5247,    49/176, -5103/18656,      0 ;
%               35/384,            0,   500/1113,   125/192,  -2187/6784, 11/84 ];
% % 4th order beta vector:
```

```
% beta4 = [ 5179/57600, 0.0, 7571/16695, 393/640, -92097/339200, 187/2100, 1/40 ];
% % 5th order beta vector:
% beta5 = [     35/384, 0.0,   500/1113, 125/192,    -2187/6784,    11/84,  0.0 ];
% % 5th order c vector:
% c = sum(alpha,2); % Sum of all rows of alpha
% % "Margin Factio" (usually 0.8-0.9)
% m = 0.8;

    case 'DOPRI87'
          c = [0 1/18 1/12 1/8 5/16 3/8 59/400 93/200 ...
                5490023248/9719169821 13/20 1201146811/1299019798 1 1]';
        A = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
         1/18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
         1/48, 1/16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
         1/32, 0, 3/32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
         5/16, 0, -75/64, 75/64, 0, 0, 0, 0, 0, 0, 0, 0, 0;
         3/80, 0, 0, 3/16, 3/20, 0, 0, 0, 0, 0, 0, 0, 0;
         29443841/614563906, 0, 0, 77736538/692538347, ...
         -28693883/1125000000, 23124283/1800000000, 0, 0, 0, 0, 0, 0, 0;
         16016141/946692911, 0, 0, 61564180/158732637, 22789713/633445777, ...
         545815736/2771057229,  -180193667/1043307555, 0, 0, 0, 0, 0, 0 ;
         39632708/573591083, 0, 0, -433636366/683701615, ...
         -421739975/2616292301, 100302831/723423059, 790204164/839813087, ...
         800635310/3783071287, 0, 0, 0, 0, 0;
         246121993/1340847787, 0 0 -37695042795/15268766246, ...
         -309121744/1061227803, -12992083/490766935, 6005943493/2108947869, ...
         393006217/1396673457, 123872331/1001029789, 0, 0, 0, 0 ;
         -1028468189/846180014, 0, 0, 8478235783/508512852, ...
         1311729495/1432422823, -10304129995/1701304382, ...
         -48777925059/3047939560, 15336726248/1032824649, ...
         -45442868181/3398467696, 3065993473/597172653, 0, 0, 0;
         185892177/718116043, 0, 0, -3185094517/667107341, ...
         -477755414/1098053517, -703635378/230739211, ...
         5731566787/1027545527, 5232866602/850066563, ...
         -4093664535/808688257, 3962137247/1805957418, ...
         65686358/487910083, 0, 0;
         403863854/491063109, 0, 0, -5068492393/434740067, ...
         -411421997/543043805, 652783627/914296604, 11173962825/925320556, ...
         -13158990841/6184727034, 3936647629/1978049680, ...
         -160528059/685178525, 248638103/1413531060, 0, 0];

    b = [14005451/335480064 0 0 0 0 -59238493/1068277825 ...
        181606767/758867731 561292985/797845732 -1041891430/1371343529 ...
        760417239/1151165299 118820643/751138087 -528747749/2220607170 1/4]';

    bhat = [13451932/455176623 0 0 0 0 -808719846/976000145 ...
            1757004468/5645159321 656045339/265891186 ...
            -3867574721/1518517206 465885868/322736535 ...
            53011238/667516719 2/45 0]';

    d = b-bhat;
    s = 13;
    otherwise
       error('Method not defined')
end

solver.AT = A';
solver.b  = b;
solver.c  = c;
solver.d  = d;
solver.stages = s;
```

```
function ESDIRKperformance(info, stats)
s = info.nStage;
I = find(stats.r > 1) % failed steps that converged
J = find(stats.Diverged > 0) % diverged steps
K = find(stats.SlowConv > 0) % slow converging steps (reached itermax)
%% Plotting (t_n, h_n) where t_{n+1} = t_n + h_n:
figure
subplot(4,2,[1 3])
semilogy(stats.t, stats.h, '.-'), hold on
semilogy(stats.t(I), stats.h(I), '.r')
semilogy(stats.t(J), stats.h(J), 'xr', 'markersize', 8, 'linewidth', 2)
semilogy(stats.t(K), stats.h(K), 'or', 'markersize', 8, 'linewidth', 2)
%xlabel('t_n')
ylabel('h_n')
%legend('t_{n+1} = t_n + h_n', 'RejStep', 'DivStep', 'SlowCon')
axis tight
%% Plotting (t_n, r_n/tol):
subplot(4,2,[5 7])
semilogy(stats.t, stats.r, '.-'), hold on
semilogy(stats.t(I), stats.r(I), '.r')
plot([min(info.tspan(1)) max(info.tspan(2))], [1 1], '-k') % steps above this line is failed steps t
xlabel('t_n')
ylabel('r_n/tol')
%legend('t_{n+1} = t_n + h_n','RejStep')
axis tight
%% Plotting (t_n, iter_n) at each stage:
for i = 1:s-1
    if i > 3, break, end
    J = find(stats.Diverged == i+1); % failed steps that converged
    K = find(stats.SlowConv == i+1); % slow converging steps (reached itermax)
    subplot(4,2,2*i) % iterations in stage i+1
    plot(stats.t, stats.iter(:, i+1), '.-b'), hold on
    plot(stats.t(I), stats.iter(I, i+1), '.r')
    plot(stats.t(J), stats.iter(J, i+1), 'xr', 'markersize', 8, 'linewidth', 2)
    plot(stats.t(K), stats.iter(K, i+1), 'or', 'markersize', 8, 'linewidth', 2)
    if i == s-1, xlabel('t_n'), end
    ylabel(['s_',int2str(i+1)])%,': ',int2str(sum(stats.iter(:, i+1), 1))])
    ylim([0 info.iterMax])
    %axis tight
end
%% Adding text:
g{1} = ['nStep : ',int2str(info.nStep)];
g{2} = ['nFail : ',int2str(info.nFail)];
g{3} = ['nDiv  : ',int2str(info.nDiverge)];
g{4} = ['nSlow : ',int2str(info.nSlowConv)];
h{1} = ['nFun  : ',int2str(info.nFun)];
h{2} = ['nJac  : ',int2str(info.nJac)];
h{3} = ['nLU   : ',int2str(info.nLU)];
h{4} = ['nBack : ',int2str(info.nBack)];
e{1} = ['Method : ',info.Method];
e{2} = ['absTol : ',num2str(info.absTol)];
e{3} = ['relTol : ',num2str(info.relTol)];
for i = 1:s-1
    f{i} = ['nIter_',int2str(i+1),' : ',int2str(sum(stats.iter(:, i+1), 1))];
end
subplot(4,2,8)
%plot(abs(stats.eig(:,1)),'x-k'), hold on, plot(abs(stats.eig(:,2)),'o-k')
axis off
ha(4) = text(-0.1,  0.7, e); ha(3) = text(0.6,  0.7, f);
ha(1) = text(-0.1, -0.4, g); ha(2) = text(0.6, -0.4, h);
set(ha, 'fontname', 'courier');
```

```
%maximize % maximizes figure window

function ESDIRKopt = ESDIRKoptions(ESDIRKsolver)

function [dfdx,dgdx]=VanderPolJac(t,x,mu)

dfdx = [0 1;-2*mu*x(1)*x(2)-1 mu*(1-x(1)*x(1))];
dgdx = eye(2);
```