

DTU Compute

Department of Applied Mathematics and Computer Science

Take-home Exam 2022

02686 Scientific Computing for Differential Equations

Sara Húnfjörð Jósepsdóttir (s212952)

Kongens Lyngby 2022



DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Preface

The code in this exam assignment was prepared in collaboration with Helga Þórey Björnsdóttir (s213615) and Olgeir Ingi Árnason (s212564).

Kongens Lyngby, May 28, 2022

Sara Húmfjörð Jósepsdóttir (s212952)

Contents

Preface	i
Contents	iii
1 Test Equation for ODEs	1
1.1 Local and Global Truncation Error	1
1.1.1 Error Analysis	2
1.2 Stability	4
1.2.1 Stability of the Euler Methods and Classical Runge Kutta	6
2 Explicit ODE solver	9
2.1 The Explicit Euler Method	9
2.1.1 Explicit Euler Fixed Step Size	10
2.1.2 Explicit Euler Adaptive Time Step and Error Estimation	10
2.1.3 Van der Pol Problem	12
2.1.4 Explicit Euler with Fixed Step Size	12
2.1.5 Explicit Euler with Adaptive Step Size and Error Estimation	14
2.1.6 Comparing with Matlab's ODE Solvers	15
3 Implicit ODE Solver	19
3.1 The Implicit Euler with Fixed Step Size	19
3.2 The Implicit Euler with Adaptive Step Size	21
3.3 Testing the Implicit Euler Method on the Van der Pol Problem	23
3.3.1 Comparing with Matlab's ODE Solvers	26
4 Solvers for SDEs	31
4.1 Multivariate Standard Wiener Process	31
4.2 The Explicit-Explicit Method	32
4.3 The Implicit-Explicit Method	33
4.4 SDE Version of the Van der Pol Problem	35
4.4.1 The SDE Methods on the Van Der Pol Problem	35
5 Classical Runge-Kutta Method	41
5.1 The Classical Runge-Kutta	41
5.2 Classical Runge Kutta with Fixed Step Size	42

5.2.1	Classical Runge-Kutta with Adaptive Step Size	43
5.3	Testing the Classical Runge-Kutta on the Van der Pol Problem	46
5.3.1	Classical Runge-Kutta with Fixed Step Size	46
5.3.2	Classical Runge-Kutta with Adaptive Step Size	48
5.3.3	Comparing with Matlab's ODE Solvers	50
6	Dormand-Prince 5(4)	53
6.1	DOPRI54	53
6.1.1	DOPRI54 with Adaptive Step Size	54
6.1.2	The Test Equation	56
6.1.3	The Van der Pol Problem	58
6.1.4	Adiabatic CSTR	60
6.1.5	Comparing with Matlab's ODE Solvers	62
7	ESDIRK23	65
7.1	ESDIRK23	65
7.1.1	Stability	67
7.1.2	ESDIRK23 with Variable Step Size	68
7.1.3	The Van der Pol Problem	70
7.1.4	Comparing with Matlab's ODE Solvers and Implemented Solvers	72
8	Discussion and Conclusion	77
A	An Appendix	79
A.1	Algorithms	79
A.1.1	Problem 2 Algorithms	79
A.1.2	Problem 3 Algorithms	81
A.1.3	Problem 4 Algorithms	84
A.1.4	Problem 5 Algorithms	86
A.1.5	Problem 6 Algorithms	89
A.1.6	Problem 7 Algorithms	94
A.2	Drivers	100
A.2.1	Problem 1 Driver	100
A.2.2	Problem 2 Driver	107
A.2.3	Problem 3 Driver	113
A.2.4	Problem 4 Driver	118
A.2.5	Problem 5 Driver	121
A.2.6	Problem 6 Driver	126
A.2.7	Problem 7 Driver	134
Bibliography		143

CHAPTER 1

Test Equation for ODEs

In this chapter the test equation is considered:

$$\dot{x} = \lambda x(t), \quad x(0) = x_0, \quad (1.1)$$

with $\lambda = -1$ and $x_0 = 1$. The analytical solution to the test equation is:

$$x(t) = e^{\lambda t} x_0 = e^{\lambda t}. \quad (1.2)$$

1.1 Local and Global Truncation Error

The *local truncation error* is a measure of the error made on each step [1]. It refers to the terms discarded when generating a numerical method from something such as a Taylor expansion (i.e. Euler's method) [2]. It is given by:

$$l_n = x_n - x_{n-1}(t_n), \quad (1.3)$$

where x_n is the numerical solution and $x_{n-1}(t_n)$ is the analytical solution starting from $x(t_{n-1}) = x_{n-1}$. The *global truncation error* is the accumulation of the *local truncation error* over all iterations, assuming perfect knowledge of the true solution at the initial time step. It is given by:

$$e_n = x_n - x(t_n), \quad (1.4)$$

where x_n is the numerical solution and $x(t_n)$ is the analytical solution at each time t_n . The local and global errors for the test equation are then given by:

$$l_n = x_n + \exp(\lambda(t_n - t_{n-1}))x_n \quad (1.5)$$

$$e_n = x_n + \exp(\lambda(t_n - t_0))x_0 \quad (1.6)$$

1.1.1 Error Analysis

The local and global errors were computed for the test equation given in (1.1) when solved with the explicit Euler method, implicit Euler method and the classical Runge-Kutta method. The driver for this test can be found in Appendix A.2.1. This was done for a fixed number of steps $N = 100$ and time interval $t = [0, 10]$. The Euclidean norm of the errors was then computed for comparison, see Table 1.1.

Table 1.1 shows that the local and global error is lowest for the classical Runge-Kutta method and highest for the explicit Euler. However, this is only for a small number of steps. To investigate the effect of the step sizes on the performance of the methods, the local and global errors were plotted against step sizes in the interval $h = [10^{-3}, 10^{-1}]$ for the three numerical methods at $t = 1$. Figure 1.1 and Figure 1.2 show that as the step size decreases the error decreases, with the classical Runge-Kutta outperforming the Euler methods for all step sizes. However, it is worth noting that for larger step sizes the difference in error decreases.

	Explicit Euler	Implicit Euler	Classical Runge-Kutta
$\ \mathbf{e}\ _2$	0.0815	0.0769	$1.43 * 10^{-6}$
$\ \mathbf{l}\ _2$	0.0111	0.0102	$1.93 * 10^{-7}$

Table 1.1: Comparison of local and global truncation errors using different ODE solvers to solve the test equation.

The accuracy of a numerical method is said to be of order p if $l_n = \mathcal{O}(h_n^{p+1})$. The order of the method thus gives an idea of how the local truncation error depends on the step size. The explicit and implicit Euler methods are of order $p = 1$ as the local truncation error is $\mathcal{O}(h^2)$. As for the classical Runge-Kutta method, it is of order $p = 4$ as the local truncation error is $\mathcal{O}(h^5)$.

Figure 1.1 shows that the curves for explicit and implicit Euler grow approximately by h^2 (as the slope is around 2) and so the complexity is $\mathcal{O}(h^2)$. From the considerations in the aforementioned paragraph this was expected as the order of the method is then $p = 2 - 1 = 1$. The same goes for the classical Runge-Kutta method, the curve grows approximately by h^5 which fits with the order of the method $p = 5 - 1 = 4$. If the local error is of complexity $\mathcal{O}(h^{p+1})$ then the complexity of the global error should be $\mathcal{O}(h^p)$ and as can be seen in Figure 1.2 the slope of the curves for classical Runge-Kutta grows by h^4 and by h^1 for the Euler methods which matches their respective order.

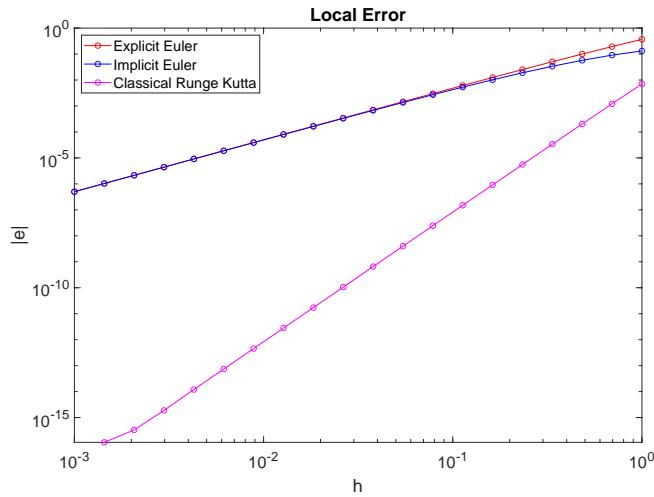


Figure 1.1: A log-log plot of the local truncation error, $|e|$, vs. step size, h , using explicit Euler, implicit Euler and the classical Runge-Kutta methods with a fixed step size on the test equation.

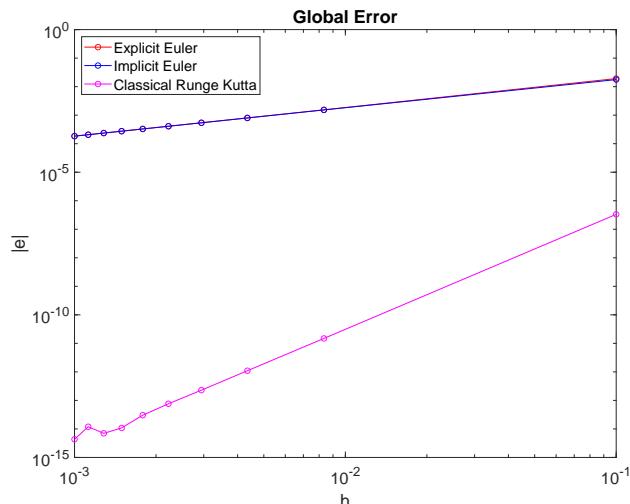


Figure 1.2: A log-log plot of global truncation error, $|e|$ vs. step size, h , using the explicit Euler, implicit Euler and the classical Runge-Kutta methods with a fixed step size on the test equation.

1.2 Stability

Convergence is not always guaranteed despite the method being consistent. This is dependent on a term called *stability*. A numerical method is said to be *stable* if small perturbations in the input produce small perturbations in the output. As mentioned in the beginning of Chapter 1, the analytical solution to the test equation is $x(t) = e^{\lambda t}x(0)$ for $t \geq 0$. Now introducing a perturbed initial condition $x_\delta = x_0 + \delta$ the analytical solution will still be $x_\delta(t) = x(0)e^{\lambda t}$, although the initial condition now implies:

$$x_\delta = (x(0) + \delta)e^{\lambda t}. \quad (1.7)$$

Given $\Re(\lambda) \leq 0$ then small perturbations in the initial condition causes only small perturbations in the solution and the problem is *stable*. On the other hand, if $\Re(\lambda) > 0$, then independent of how small the perturbation of the initial conditions are, there will be large changes in the solution. This type of problem is called *unstable*. If $\Re(\lambda) > 0$ the problem is considered *asymptotically stable*. The linear stability domain of a numerical method \mathcal{D} is the set of $z = \lambda h$ ($h = \Delta t$) for which the growth factor is less than one, given by:

$$\mathcal{D} = \{z \in \mathbb{C} : |R(z)| < 1\} \quad (1.8)$$

where $R(z)$ is the transfer function of the numerical method for the test equation, that is $x_{n+1} = R(z)x_n$. Inserting the test equation, (1.1), to one step of the explicit Euler method:

$$\begin{aligned} x_{n+1} &= x_n + h\lambda x_n \\ \Rightarrow x_{n+1} &= (1 + \lambda h)x_n, \end{aligned}$$

then by induction:

$$x_n = (1 + \lambda h)^n x_0.$$

For $\lambda < 0$ the analytical solution is exponentially decaying and a numerical method should exhibit the same behaviour in order to be stable. That is the limit of the approximations of x_n should go towards 0 as n goes towards infinity. The stability condition for the explicit Euler is therefore:

$$\mathcal{D}_{\text{explicitEuler}} = \{z = \lambda h \in \mathbb{C} : |1 + z| < 1\} \quad (1.9)$$

Now for a step of the implicit Euler one obtains:

$$\begin{aligned}
x_{n+1} &= x_n + h\lambda x_{n+1} \\
\Rightarrow x_{n+1} &= \frac{1}{1 - \lambda h} x_n \\
\Rightarrow x_n &= \left(\frac{1}{1 - \lambda h} \right)^n x_0.
\end{aligned}$$

The stability condition for the implicit Euler is therefore:

$$\mathcal{D}_{implicitEuler} = \{z = \lambda h \in \mathbb{C} : \left| \frac{1}{1 - z} \right| < 1\} \quad (1.10)$$

Now for the classical Runge-Kutta method. Applying a general Runge-Kutta to the test equation, the relations for the internal stages, s , are given by:

$$X_i = x_n + z \sum_{j=1}^s a_{ij} X_j, \quad i = 1, \dots, s. \quad (1.11)$$

The above system can be cast into matrix form by introducing the vector $e = (1, \dots, 1)^T \in \mathbb{R}^s$:

$$\begin{aligned}
X &= x_n \mathbf{1} - z A Y, \\
\Rightarrow X &= x_n (I - z A)^{-1} e,
\end{aligned} \quad (1.12)$$

where $X = (X_1, \dots, X_s)^T$ and A is a matrix with entries a_{ij} , now writing it in another form:

$$x_{n+1} = x_n + z \sum_{j=1}^s b_j X_j = x_n + z b^T X, \quad (1.13)$$

and combining (1.12) and (1.13) the desired stability transfer function, $R(z) = 1 + z b^T (I - z A)^{-1} e$, can be obtained. For the classical Runge Kutta with $s = 4$ internal stages the stability region can then be defined as:

$$\mathcal{D}_{classicalRK} = \{z = \lambda h \in \mathbb{C} : |1 + z b^T (I - z A)^{-1} e| < 1\}. \quad (1.14)$$

The definition of A-stability is such, that the entire left half plane is contained within the stability region. It has the property that the origin is stable regardless of the step size. In mathematical terms:

$$|R(z)| < 1, \quad \forall z : \Re e(z) < 0. \quad (1.15)$$

A method is said to be L-stable if it is A-stable and the following holds:

$$\lim_{z \rightarrow -\infty} |\mathcal{R}(z)| = 0. \quad (1.16)$$

The above derivations are heavily based on Chapter 4 from lecture notes in [3], Chapter 10 in the lecture notes from [4], and lecture 4B on "Order and Stability for some Simple Methods" [5].

1.2.1 Stability of the Euler Methods and Classical Runge Kutta

Figure 1.3 and Figure 1.4 show the stability regions for the implicit and the explicit Euler method. For the implicit Euler method it can be seen the stability region is the exterior of the disk of radius 1 centered at 1 in the complex plane, with $\max(R(z)) = 0.99$. However, for the explicit Euler $\max(R(z)) = 6.403$ and it can be seen the stability region is the entire outer yellow region in the figure. From this it is evident that the condition for A-stability is fulfilled by the implicit Euler method, but not for the explicit Euler method. As for L-stability, for the explicit Euler the limit $\lim_{z \rightarrow -\infty} |\mathcal{R}(z)| \rightarrow \infty$ so it is not L-stable, it's stability is dependent on the choice of the step size. On the other hand $\lim_{z \rightarrow -\infty} |\mathcal{R}(z)| = 0$ for the implicit Euler and it is therefore L-stable.

Figure 1.5 shows the stability region for the classical Runge Kutta method. Evidently, the stability region does not include the entire left-half plane. and $\max(R(z)) = 69.04$ which confirms this method is not A-stable. The condition for L-stability is also not satisfied as $\lim_{z \rightarrow -\infty} |\mathcal{R}(z)| \rightarrow \infty$. To conclude, for the classical Runge-Kutta and explicit Euler the origins are only stable provided h is suitably restricted while the implicit Euler is unconditionally stable.

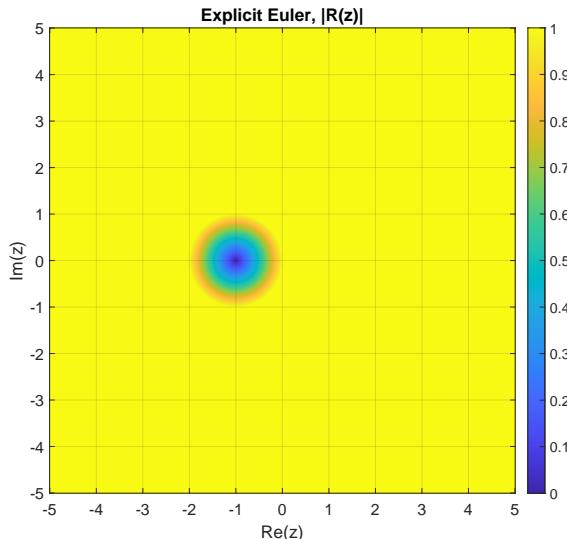


Figure 1.3: Stability region of the explicit Euler method.

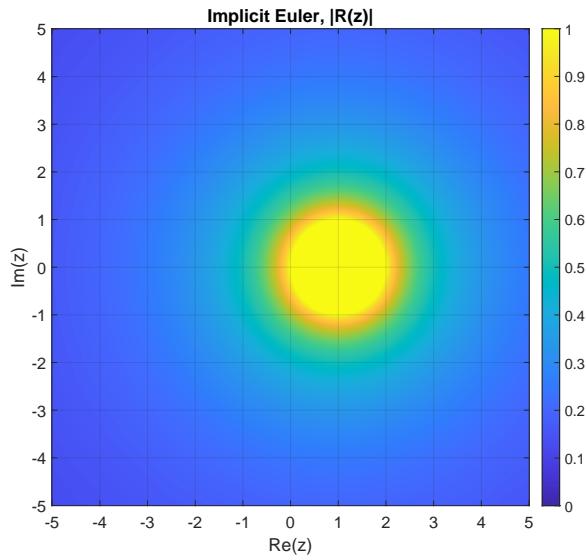


Figure 1.4: Stability region of the implicit Euler method.

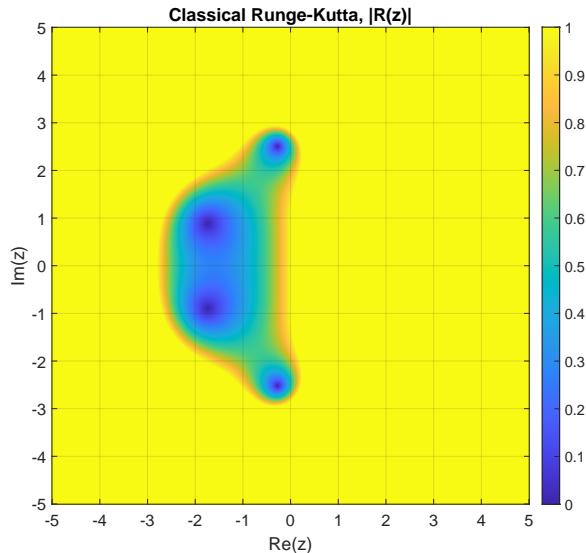


Figure 1.5: Stability region of the classical Runge-Kutta method.

CHAPTER 2

Explicit ODE solver

In this chapter the following initial value problem is considered:

$$\dot{x}(t) = f(t, x(t), p), \quad x(t_0) = x_0, \quad (2.1)$$

where $x \in \mathbb{R}^{n_x}$ and $p \in \mathbb{R}^{n_p}$. Henceforth f is assumed to have sufficient smoothness so as to guarantee a unique existence of a solution $x(t)$. In the preceding sections, the explicit Euler method is described and its implementations tested on the Van der Pol problem. The performances of the methods are then compared to Matlab's own ODE solvers, *ode45* and *ode15s*. The driver for this chapter can be found in Appendix A.2.2.

2.1 The Explicit Euler Method

The explicit Euler algorithm is the simplest and most intuitive numerical method for solving initial value problems.

To approximate the solution to (2.1) the interval of integration, $[t_0, t_N]$, can be discretized by a mesh:

$$t_0 < t_1 < \dots < t_{N-1} < t_N.$$

Now defining the n th step size as $h_n = t_n - t_{n-1}$, the approximate values can be constructed as:

$$x_0, x_1, \dots, x_{N-1}, x_N,$$

with x_n being an intended approximation of $x(t_n)$. Now concentrating on one step, $n \geq 1$ and considering Taylor's expansion:

$$x(t_n) = x(t_{n-1}) + h_n x'(t_{n-1}) + \frac{1}{2} h_n^2 x''(t_{n-1}) + \dots,$$

which can be written in order notation as:

$$x(t_n) = x(t_{n-1}) + h_n x'(t_{n-1}) + O(h_n^2)$$

The explicit Euler method can then be derived by dropping the rightmost term in this Taylor expansion and replacing x' with f , yielding:

$$x_n = x_{n-1} + h_n f(t_{n-1}, x_{n-1}). \quad (2.2)$$

The step in Euler's algorithm amounts to taking a straight line in the tangential direction to the exact trajectory starting at (t_{n-1}, x_{n-1}) and continuing until the end of the step. The aim is to have h small enough such that x_n is not too far from $x(t_n)$. The above derivations are based on Chapter 3 in Ascher and Petzol [1].

2.1.1 Explicit Euler Fixed Step Size

The explicit Euler method with fixed step size was implemented in Matlab, the algorithm for implementing the method is presented in Algorithm 1 and the matlab code is listed in Listing A.1.

Algorithm 1: Explicit Euler with fixed step size.

Data: t_0, t_N, x_0 , number of steps N , function f
Result: t, x

```

1 Compute step size  $h = \frac{t_N - t_0}{N}$ 
2 for  $i$  from 1 to  $N$  do
3    $f_i = f(t_i, x_i)$                                 // Evaluate function at current point
4    $t_{i+1} = t_i + h$ 
5    $x_{i+1} = x_i + f_i * h$ 
6 end

```

2.1.2 Explicit Euler Adaptive Time Step and Error Estimation

Introducing an adaptive step size can help control the errors of the methods and ensure certain stability properties. The idea behind step doubling is simple. Error estimation by step doubling involves approximating solutions by taking a full step, x_k and then a half-step \hat{x}_k . The difference between these solutions then gives an estimate of the local error of the less accurate one of the two approximations [1]:

$$|x_k - \hat{x}_k| \leq TOL \quad (2.3)$$

where TOL is a specified tolerance. To control the step size, an asymptotic step size controller is used:

$$h_{k+1} = \left(\frac{\epsilon}{r_{k+1}} \right)^{\frac{1}{2}} h_k, \quad (2.4)$$

where r_{k+1} is the error estimator of the next step and ϵ is the tolerance, here specified at 0.8. Then at each iteration a new step size is computed until (2.3) is satisfied. The step is accepted and a new value approximated. This is done until the final time has been reached.

The explicit Euler method was modified to include an adaptive step size and error estimation using step doubling and the full procedure is explained in Algorithm 2. The Matlab implementation is listed in Appendix A.1.1. Stats are included in the output to keep track of function evaluations, errors and step sizes.

Algorithm 2: Explicit Euler with adaptive step size and error estimation by step doubling.

```

Data:  $t_0, t_f, x_0$ , initial step size  $h_0$ , objective function  $F$ , abstol, reltol
Result:  $t, x$ 

1 Define error controller parameters
2  $\epsilon = 0.8$  // Target
3  $\mathcal{F}_{min} = 0.1$  // Maximum decrease factor
4  $\mathcal{F}_{max} = 5.0$  // Maximum increase factor
5 while Final time  $t_f$  has not been reached do
6   if  $t + h > t_f$  then
7     |  $h = t_f - t$  // Reduce step
8   end
9    $f = F(t_0, x)$  // Evaluate function at current point
10  while Step is NOT accepted do
11    |  $x_{next} = x + hf$ 
12    | Step doubling
13    |  $h_{temp} = \frac{1}{2}h$ 
14    |  $t_{temp} = t + h_{temp}$ 
15    |  $x_{temp} = x + h_{temp}f$  // Take half a step
16    |  $f_{temp} = F(t_{temp}, x_{temp})$  // Evaluate function at half-step
17    |  $\hat{x} = x_{temp} + h_{temp} * f_{temp}$ 
18    | Error estimation
19    |  $e = \hat{x} - x_{next}$ 
20    |  $r = \max\left\{\frac{|e|}{\max\{abstol, |\hat{x}|reltol\}}\right\}$ 
21    if  $r \leq 1.0$  then
22      | Step is accepted - update point
23      |  $t := t + h$ 
24      |  $x := \hat{x}$ 
25    end
26    | Update step size using asymptotic step size controller
27    |  $h := \max\{\mathcal{F}_{min}, \min\{\sqrt{\epsilon/r}, \mathcal{F}_{max}\}\} * h$ 
28  end
29 end

```

2.1.3 Van der Pol Problem

The Van der Pol equation is a classical example of a self-oscillatory system. It is a very useful mathematical model with applications in physics and biology [6]. It is a second order differential equation defined as:

$$y''(t) = \mu(1 - y(t)^2)y'(t) - y(t). \quad (2.5)$$

The problem can be converted to a system of first order differential equations:

$$\dot{x}_1(t) = x_2(t) \quad (2.6a)$$

$$\dot{x}_2(t) = \mu(1 - x_1(t)^2)x_2(t) - x_1(t), \quad (2.6b)$$

with $y(t) = x_1(t)$. The scalar parameter μ indicates the nonlinearity and strength of the dampening and the stiffness of the problem depends on this parameter. In the preceding subsections, the explicit Euler with fixed and adaptive step size described above are tested on the Van der Pol problem. The initial point $x_0 = [1.0, 1.0]^T$ was used and the methods were tested for $\mu = 3$ and $\mu = 20$ for a time interval $t = [0, 50]$. The initial step size for the adaptive method was set to $h_0 = 0.001$ and two fixed number of steps of $N = 3000$ and $N = 10000$ were tested. The absolute and relative tolerances used were 10^{-5} . The driver for performing the tests can be found in Appendix A.2.3.

2.1.4 Explicit Euler with Fixed Step Size

The results for the explicit Euler with fixed step size are visualized in Figure 2.1. As can be seen, more than $N = 3000$ steps are required for the stiff Van der Pol ($\mu = 20$) to avoid a large error. This is due to the fact that when μ is large it can cause rapid variations in the solution. The explicit Euler with fixed step size becomes numerically unstable for stiff problems if the step size is too large. This is evident from the phase plot in Figure 2.1(b). The non-stiff system however, can easily handle smaller steps and for $h = 0.005$ the method manages to perform efficiently.

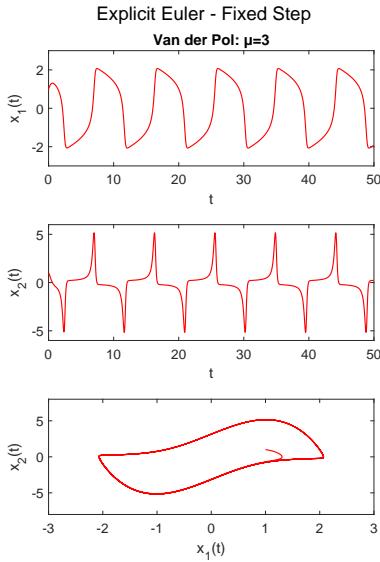
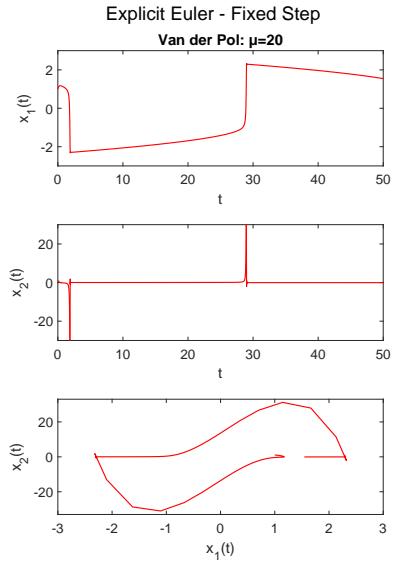
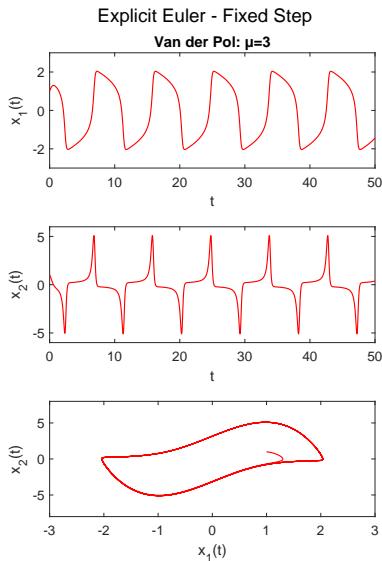
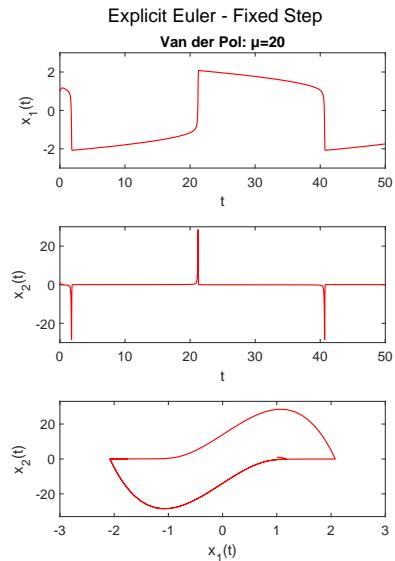
((a)) $h = 0.0167$.((b)) $h = 0.0167$.((c)) $h = 0.005$.((d)) $h = 0.005$.

Figure 2.1: Explicit Euler with fixed step size, h , on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.

2.1.5 Explicit Euler with Adaptive Step Size and Error Estimation

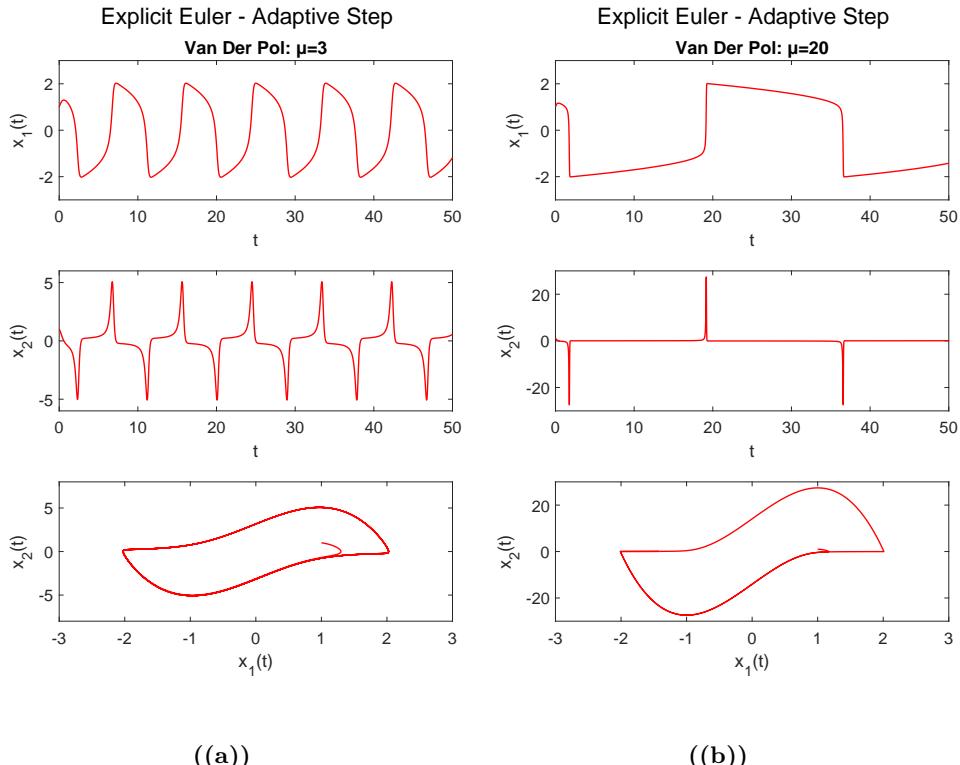


Figure 2.2: Explicit Euler with adaptive step size $h_0 = 0.001$ on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.

Figure 2.2 shows the results from using the explicit Euler method with adaptive time step and error estimation on the Van der Pol problem. An initial step size was chosen to be such that the step size would not fluctuate too much when solving the non-stiff system and was therefore set at $h = 0.001$. As can be seen the behaviour is the same as for the explicit Euler with fixed step size. In the upmost plot in Figure 2.2(b) there are regions where the solution components in the limit cycle change slowly, which shows that the problem is very stiff. Alternatively, in Figure 2.2(a) for the non-stiff system there are regions of very sharp changes.

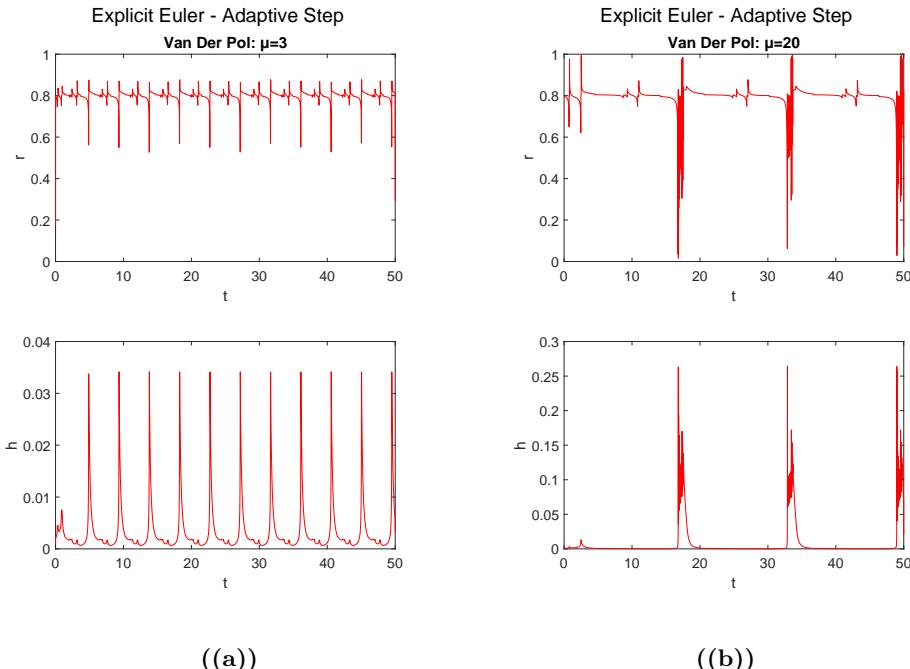


Figure 2.3: The error estimator, r , and step sizes, h using the Explicit Euler with adaptive step size on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.

Figure 2.3 show the behaviour of the step size, h , and the error estimator, r , during the explicit Euler adaptive step procedure. For the stiff system, the error estimator jumps towards the "reject step" value of 1 when the step size gets too large. The non-stiff system however has smaller fluctuations in the step size which means more steps are accepted. This is investigated further in the next section.

2.1.6 Comparing with Matlab's ODE Solvers

The explicit Euler methods were compared with Matlab's inbuilt ODE solvers *ode45* and *ode15s*. According to the Matlab documentation, *ode45* is defined as an ODE solver for nonstiff problems, it implements the explicit one-step four-stage Runge–Kutta method. On the other hand, *ode15s* is a variable-step, variable-order solver for stiff problems, based on the numerical differentiation formulas (NDFs) of orders 1 to 5.

The test was done for absolute and relative tolerances of 10^{-3} and 10^{-7} and an initial step size $h_0 = 0.001$ for time interval $t = [0, 50]$. As can be seen in Table 2.2 going down to 10^{-7} in tolerance is way too much for the explicit Euler method to handle as the function evaluations become extremely high. Going lower than this can

increase computation time by a high degree. Both *ode45* and *ode15s* work efficiently even at low tolerances, see Table 2.1, with only a slight increase in function evaluations when decreasing the tolerance.

Table 2.1: Comparison of explicit Euler with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-3} .

	Explicit Euler		ode45		ode15s	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
N. Function	3175	2142	1489	3751	1393	750
N. Steps	1658	1177	248	625	675	363
N. Accepted	1517	965	194	583	559	287
N. Rejected	141	212	54	42	116	76

Table 2.2: Comparison of explicit Euler with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-7} .

	Explicit Euler		ode45		ode15s	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
N. Function	300800	134550	6307	6769	4420	2325
N. Steps	150401	67273	683	1128	2659	1274
N. Accepted	150400	67273	656	1115	2505	1171
N. Rejected	1	2	27	13	154	103

To visualize the results, they were plotted on the same graph along with the approximation from the explicit Euler with fixed step size $h = 0.001$. This was done for both a stiff and non-stiff Van der Pol system. From Figure 2.4(a) it can be seen that for the non-stiff system at high tolerances all methods manage to perform quite well with the explicit Euler methods lagging slightly behind as time goes on. However, this difference decreases as the tolerance decreases as Figure 2.4(b) shows. For the stiff system, Figure 2.5(a) shows that the explicit Euler with adaptive step manages to perform quite efficiently at high tolerances, catching up with the *ode45* and *ode15s* solutions when the tolerance is decreased. However, with a fixed step size the approximations are not as good even at low tolerances and the step size would need to be reduced.

In conclusion, the explicit Euler with fixed step size requires a very small step size in order to perform as well as Matlab's ODE solvers. With adaptive step and error estimation the method works well for both the stiff and non-stiff system but the tolerances should be set lower than 10^{-3} for the best performance. Although, the method becomes very slow for tolerances lower than 10^{-7} .

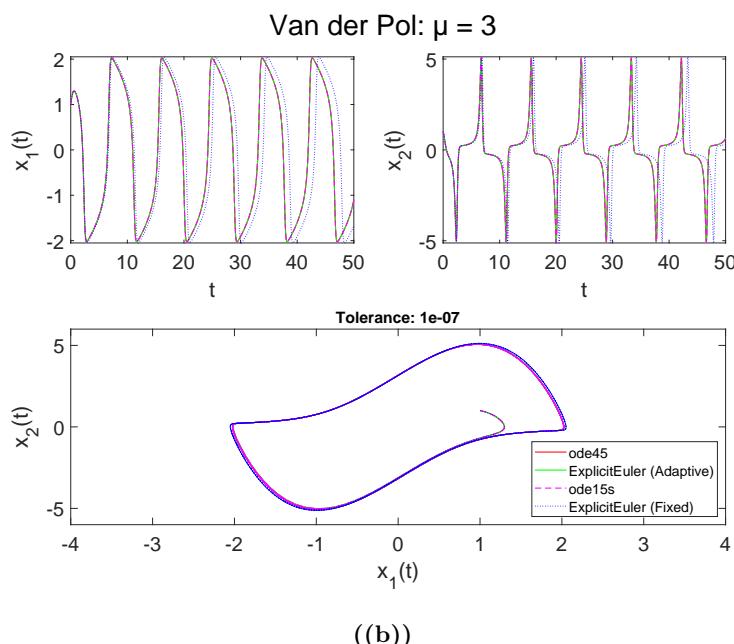
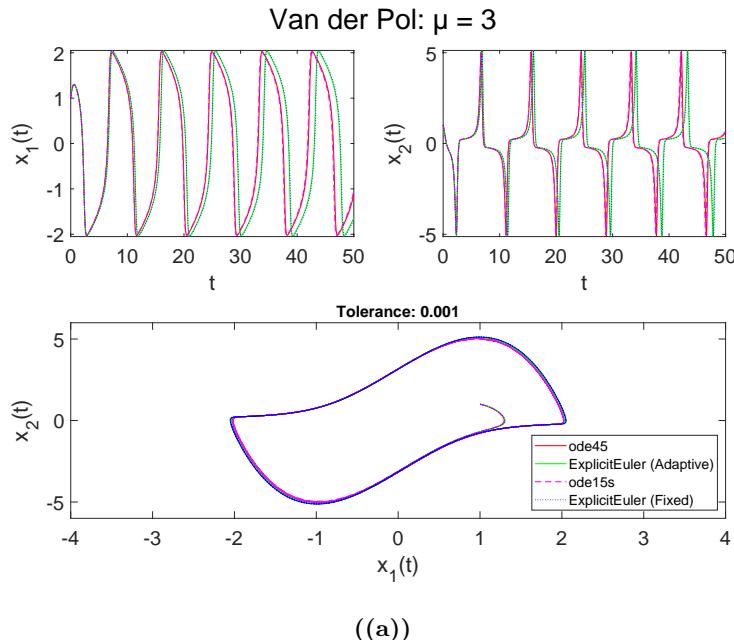


Figure 2.4: Comparison of explicit Euler with fixed and adaptive step size, *ode45* and *ode15s* on the Van der Pol problem, $\mu = 3$.

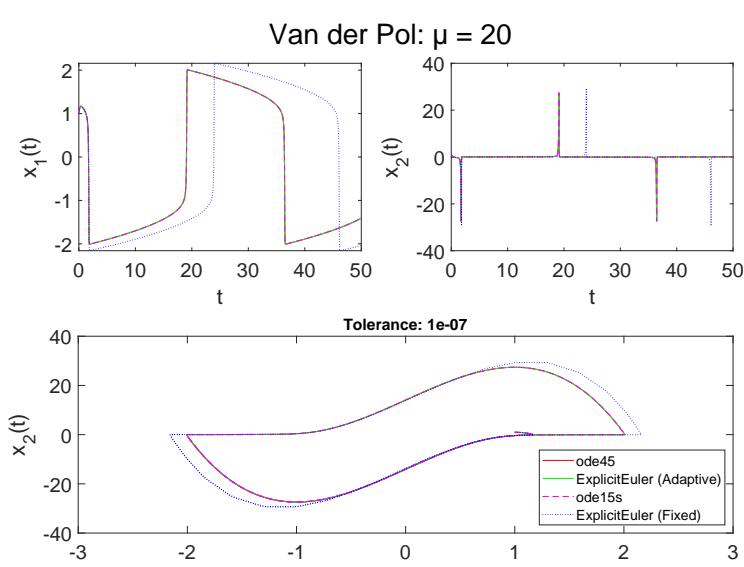
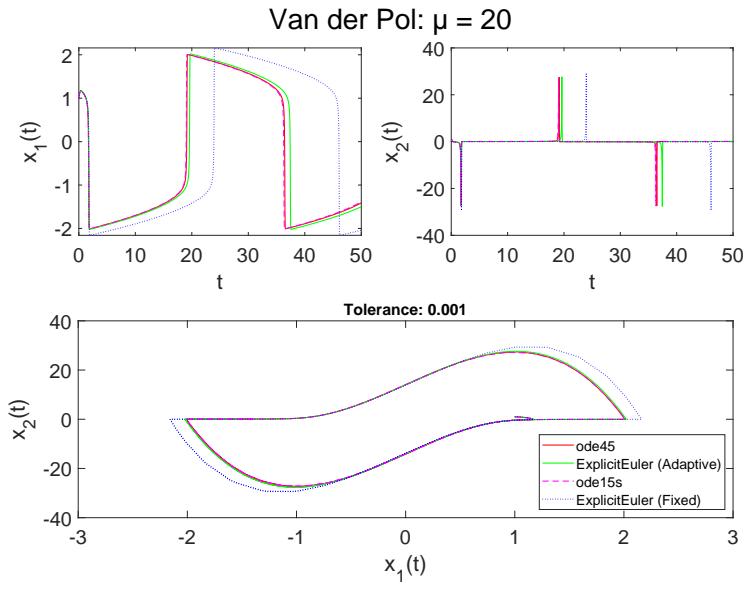


Figure 2.5: Comparison of explicit Euler with fixed and adaptive step size, *ode45* and *ode15s* on the Van der Pol problem, $\mu = 20$.

CHAPTER 3

Implicit ODE Solver

In this chapter the same initial value problem as described in (2.1) is considered. Stated here again for good measure:

$$\dot{x}(t) = f(t, x(t), p), \quad x(t_0) = x_0, \quad (3.1)$$

In the preceding sections, the implicit Euler method and it's modifications are described and the implementations tested on the Van der Pol problem. The performances of the methods are then compared to Matlab's own ODE solvers, *ode45* and *ode15s*. The driver for this chapter can be found in Appendix A.2.3.

3.1 The Implicit Euler with Fixed Step Size

Instead of using the previous iterate as in the explicit Euler method the implicit Euler method looks at future values to approximate a solution. For nonlinear problems, the solution may require the use of numerical solver's, such as Netwon's method. This makes the implicit Euler more computationally demanding than the explicit one. As for the explicit Euler there are a variety of ways to derive the implicit Euler. One such is explained in Lecture 5 on "Stiff Systems and the Implicit Euler Method" [5] where the integral:

$$\int_{t_k}^{t_{k+1}} f(t, x(t)) dt \quad (3.2)$$

can be approximated using the right-side-evaluation, which gives:

$$x_{k+1} = x_k + (t_{k+1} - t_k) f(t_{k+1}, x_{k+1}) \quad (3.3)$$

Now letting $\Delta t = t_{k+1} - t_k$ then the step in the implicit Euler method is obtained:

$$x_{k+1} = x_k + \Delta t f(t_{k+1}, x_{k+1}) \quad (3.4)$$

Now, x_{k+1} can be approximated by the use of Newton's method by solving the residual term:

$$R(x_{k+1}) = x_{k+1} - \Delta t f(t_{k+1}, x_{k+1}) - x_k = 0 \quad (3.5)$$

To show how that is done first consider a first order Taylor expansion of R around x_k and set it to 0:

$$R(x) \approx R(x_k) + \frac{\partial R}{\partial x}(x_k)(x - x_k) = 0 \quad (3.6)$$

Newton's method is:

$$R(x_{k+1}) \approx R(x_k) + \frac{\partial R}{\partial x}(x_k)(x_{k+1} - x_k) = 0 \quad (3.7)$$

$$x_{k+1} = x_k + \nabla x \quad (3.8)$$

Now, defining $b := R(x_k)$ and $A := \frac{\partial R}{\partial x}(x_k)$ the above problem becomes as easy as solving $A\nabla x = b$ for ∇x and then the next step is computed as:

$$x_{k+1} = x_k - \nabla x \quad (3.9)$$

The algorithm to implement the implicit Euler with fixed step size can be seen in Algorithm 3 along with the Newton step in Algorithm 4. The Matlab code for this method can be found in Appendix A.1.2.

Algorithm 3: Implicit Euler with fixed step size.

Data: t_0, t_N, x_0 , number of steps N , objective function F
Result: t, x

```

1 Set  $\epsilon = 10^{-8}$                                 // Tolerance for Newton's method
2 Compute step size  $h = \frac{t_N - t_0}{N}$ 
3 Set  $x_1 = x_0$ 
4 for  $i$  from 1 to  $N$  do
5    $[f_i, J] = F(t_i, x_i)$                       // Evaluate function at current point
6    $t_{i+1} = t_i + h$ 
7    $x_{init} = x_i + f_i * h$ 
8    $\hat{x}_i = \text{NewtonsMethodODE}(t_i, x_i, x_{init}, h, F, \epsilon)$  // Approximate next step
9   Update point
10   $x_{i+1} := \hat{x}_i + f_i * h$ 
11 end

```

Algorithm 4: Newton's method to approximate the next step in the implicit Euler method.

```

1 Procedure NewtonsMethodODE
  Data:  $t_i, x_i, x_{init}$ , step size  $h$ , objective function  $f, \epsilon$ 
  Result:  $x$ 
  2 Set  $t = t_i + h$ 
  3 Set  $x = x_{init}$ 
  4  $[f, J] = F(t, x)$                                 // Evaluate function at current point
  5 Set  $x = x_{init}$ 
  6 Set  $R = x - f_i * h - x_i$                       // Initialize residual term
  7 while  $\|R\|_2 > \epsilon$  do
    8  $\frac{\partial R}{\partial x} = I - J * h$ 
    9  $\nabla x = R / \frac{\partial R}{\partial x}$ 
    10  $x := x - \nabla x$ 
    11  $[f, J] = F(t, x)$                                 // Evaluate function at current point
    12  $R := x - f * h - x_i$                           // Update residual term
  13 end

```

3.2 The Implicit Euler with Adaptive Step Size

The implicit Euler method can also be modified to include error estimation by step doubling and an adaptive step size controller. The procedure is the same as described in Section 2.1.2 but now with the addition of the Newton step at each step. The algorithm to implement this method can be seen in Algorithm 5 with the Newton step described above in Algorithm 4. The Matlab code for this method can be found in Appendix A.1.2.

Algorithm 5: Implicit Euler with adaptive step size and error estimation.

Data: t_0, t_f, x_0 , initial step size h_0 , objective function F , abstol,reltol
Result: t, x

```

1 Set  $\epsilon = 0.8$                                 // Tolerance
2 Set  $\mathcal{F}_{min} = 0.1$                       // Maximum decrease factor
3 Set  $\mathcal{F}_{max} = 5.0$                       // Maximum increase factor
4 Set  $h = h_0$ 
5 Set  $x = x_0$ 
6 while Final time  $t_f$  has not been reached do
7   if  $t + h > t_f$  then
8     |  $h = t_f - t$                                 // Reduce step
9   end
10   $f = F(t_i, x_i)$                           // Evaluate function at current point
11  while Step is NOT accepted do
12     $x_{init} = x + f * h$ 
13     $x_{next} = \text{NewtonsMethodODE}(t, x, x_{init}, h, F, \epsilon)$  // Approximate next
        step
14    Step doubling
15     $h_{temp} = 0.5h$ 
16     $t_{temp} = t + h_{temp}$ 
17     $\hat{x}_{init} = x + fh_{temp}$                   // Take half a step
18     $x_{temp} = \text{NewtonsMethodODE}(t, x, \hat{x}_{init}, h_{temp}, F, \epsilon)$ 
19     $f_{temp} = F(t_{temp}, x_{temp})$             // Evaluate function
20     $\hat{x} = x_{temp} + h_{temp} * f_{temp}$ 
21    Error estimation
22     $e = x_{next} - \hat{x}$ 
23     $r = \max\left\{\frac{|e|}{\max\{abstol, |\hat{x}| reltol\}}\right\}$ 
24    if  $r \leq 1.0$  then
25      | Step is accepted - update point
26      |  $t := t + h$ 
27      |  $x := \hat{x}$ 
28    end
29    Update step size using asymptotic step size controller
30     $h := \max\{\mathcal{F}_{min}, \min\{\sqrt{\epsilon/r}, \mathcal{F}_{max}\}\}h$ 
31  end
32 end

```

3.3 Testing the Implicit Euler Method on the Van der Pol Problem

The implicit Euler methods described in the previous sections were tested on the Van der Pol problem described in (2.6). The driver for performing the tests can be found in Appendix A.2.3. The same setup was used as described in Section 2.1.3.

Figure 3.1 shows the results for using the implicit Euler with fixed step size on the Van der Pol problem in (2.6). Apparently, the implicit Euler does a better job of approximating the stiff system for smaller step size than the explicit Euler above, see Figure 3.1(b). According to Chapter 3 in the book by Ascher and Petzold [1] this makes sense as in general the implicit Euler requires fewer steps than the explicit for stiff problems. Additionally, the implicit Euler has what is called "stiff decay" which means the method is able to skip rapidly varying solution details and still obtain a descent solution for very stiff systems [1]. This is noticeable in the smoother fluctuations in Figure 3.1(b) than in Figure 2.1(b) for the explicit Euler method. For the non-stiff system in Figure 3.1(a) and Figure 3.1(c) the difference in the output is considerably small which shows that the method performs efficiently even for smaller steps.

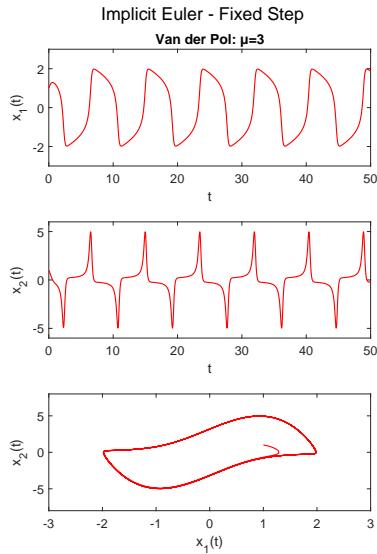
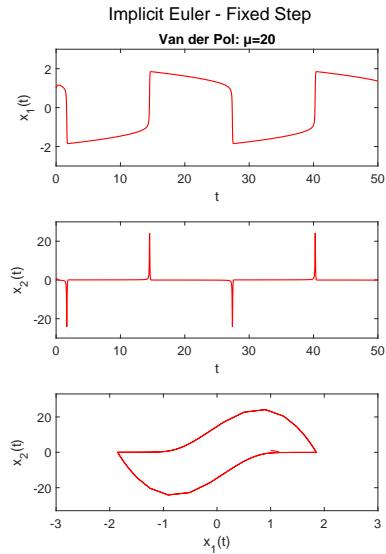
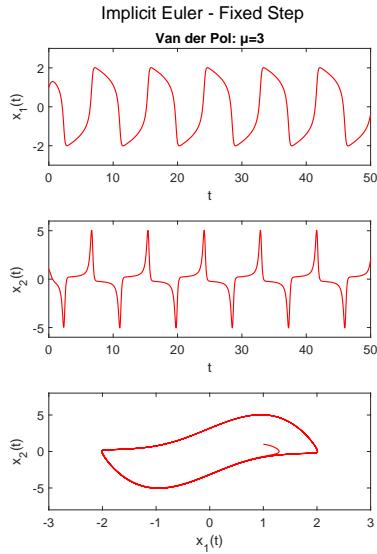
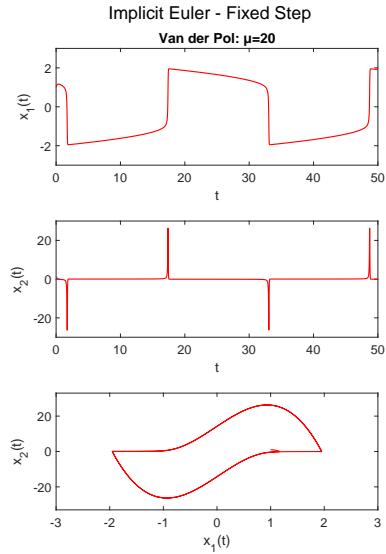
((a)) $h = 0.0167$.((b)) $h = 0.0167$.((c)) $h = 0.005$.((d)) $h = 0.005$.

Figure 3.1: Implicit Euler with fixed step size, h , on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.

Moving on to the results for the implicit Euler with adaptive steps. The results are visualized in Figure 3.2 and as can be seen the method performs well for both the stiff and non-stiff system. Investigating further, Figure 3.3 shows the step sizes, h , and error estimator, r , for the time span. Figure 3.3(b) clearly shows that the method is accepting the steps despite high fluctuations in contrast to the explicit Euler solution. This is especially true for the stiff system, see Figure 3.2(b). To conclude, the implicit Euler works well for stiff systems for reasons described above, outperforming the explicit Euler in this regard. Both methods however seem to have similar performances for non-stiff systems and perform quite efficiently.

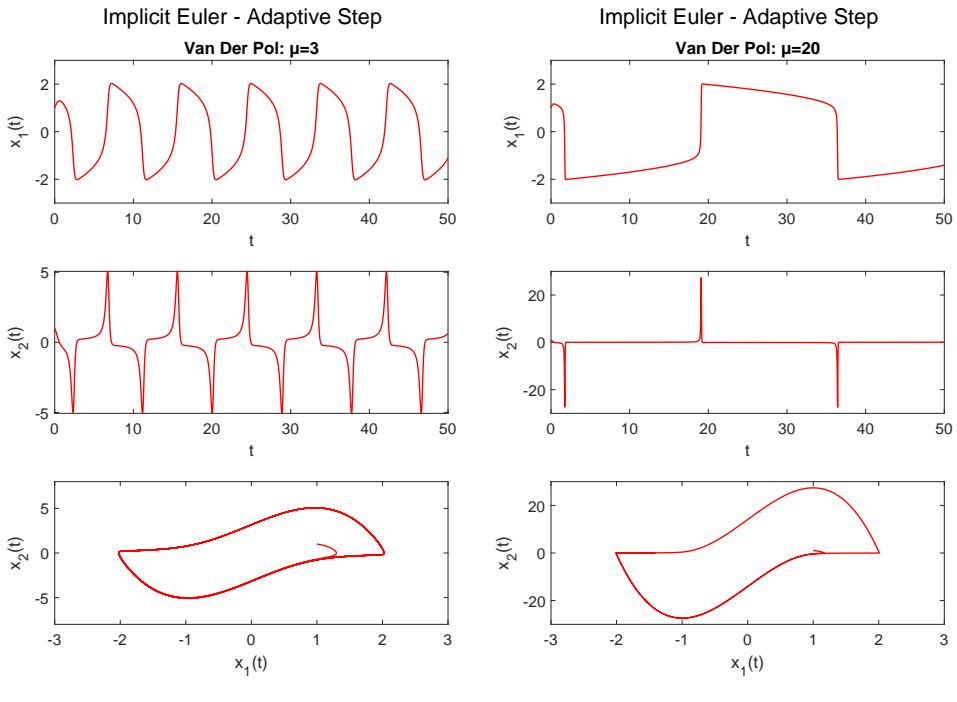
((a)) $\mu = 3$.((b)) $\mu = 20$.

Figure 3.2: Implicit Euler with adaptive step size $h_0 = 0.001$ on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.

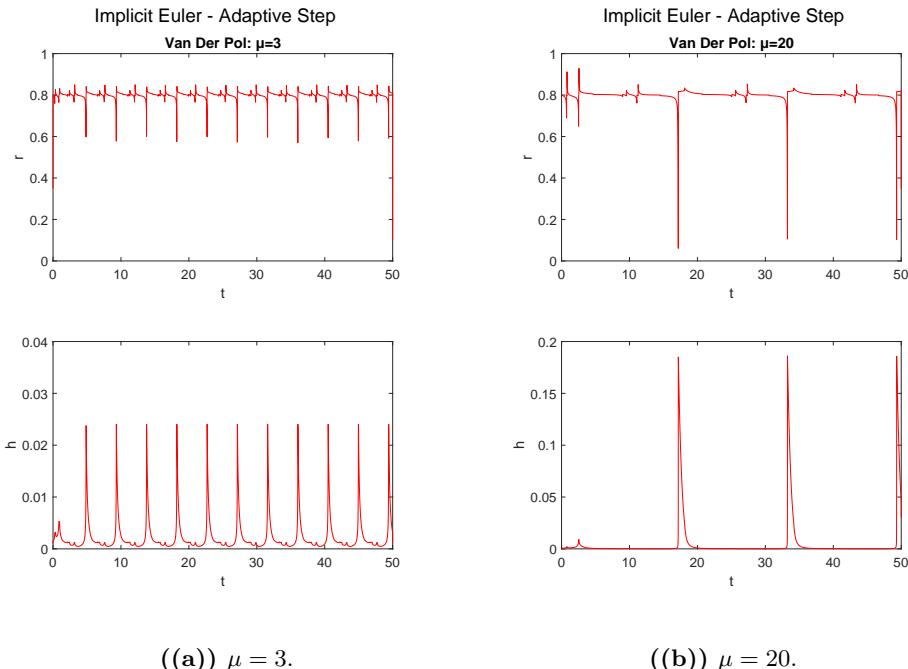


Figure 3.3: Error estimation and step sizes using the implicit Euler with adaptive step size on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.

3.3.1 Comparing with Matlab's ODE Solvers

In this section, the performance of the implicit Euler method was further tested by comparing it with Matlabs ODE solvers, *ode45* and *ode15s*. The same setup was used as in Section 2.1.6.

Table 3.1 shows a comparison of the solver statistics for absolute and relative tolerance of 10^{-3} . Compared to the ODE solvers, the implicit Euler with adaptive steps takes considerably more steps for the non-stiff system. Decreasing the tolerance increases the steps and function evaluations slightly for the ODE solvers, see Table 3.2. On the other hand, for the implicit Euler then going from 10^{-3} to 10^{-7} increases the function evaluations dramatically, reaching extremely high values.

Table 3.1: Comparison of implicit Euler with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-3} .

	Implicit Euler		ode45		ode15s	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
N. Function	8713	4069	1489	3751	1393	750
N. Steps	2200	1037	248	625	675	363
N. Accepted	2113	958	194	583	559	287
N. Rejected	87	79	54	42	116	76

Table 3.2: Comparison of implicit Euler with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-7} .

	Implicit Euler		ode45		ode15s	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
N. Function	850730	380480	6307	6769	4420	2325
N. Steps	212681	95121	1051	1128	2659	1274
N. Accepted	212680	95119	1024	1115	2505	1171
N. Rejected	1	2	27	13	154	103

To visualize the results, they were plotted on the same graph along with the approximation from the implicit Euler with fixed step size $h = 0.001$. This was done for both a stiff and non-stiff Van der Pol system. From Figure 3.4 it can be seen the implicit Euler with adaptive steps manages to achieve sufficient smoothness even at low tolerances, with the ODE solvers agreeing on the same solution. This is also apparent in Figure 3.5. On the other hand, with fixed steps, the method starts to lag a bit behind after $t = 20$ for the non-stiff system and does not match the other solutions for the stiff system. To conclude, the tests manage to demonstrate that the implicit Euler with adaptive steps converges as expected. It manages to accurately solve the Van der Pol problem, even at low tolerances. The disadvantage is that for low tolerances the method becomes slow as the number of required steps increases.

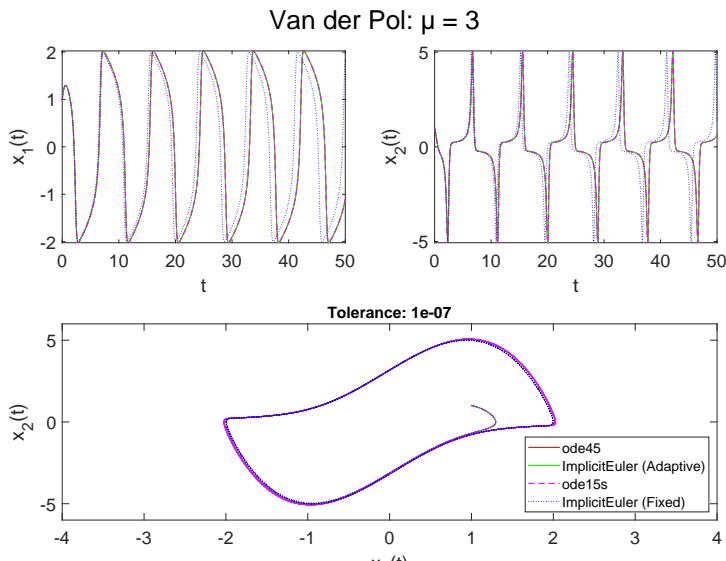
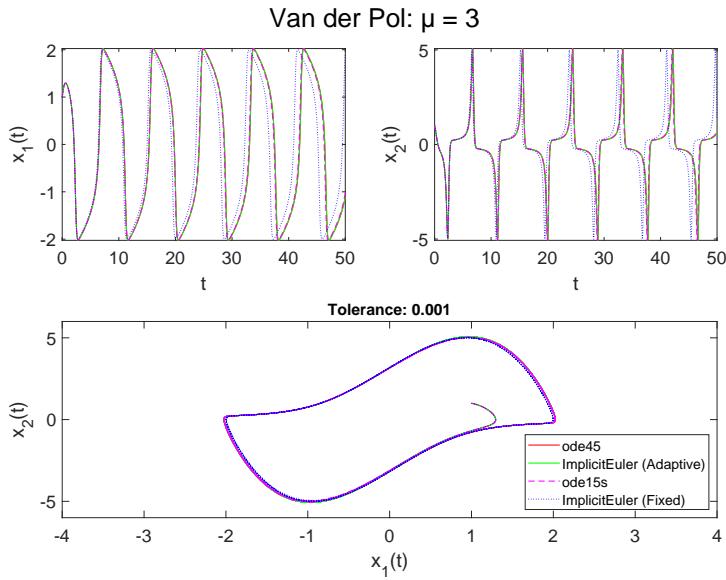
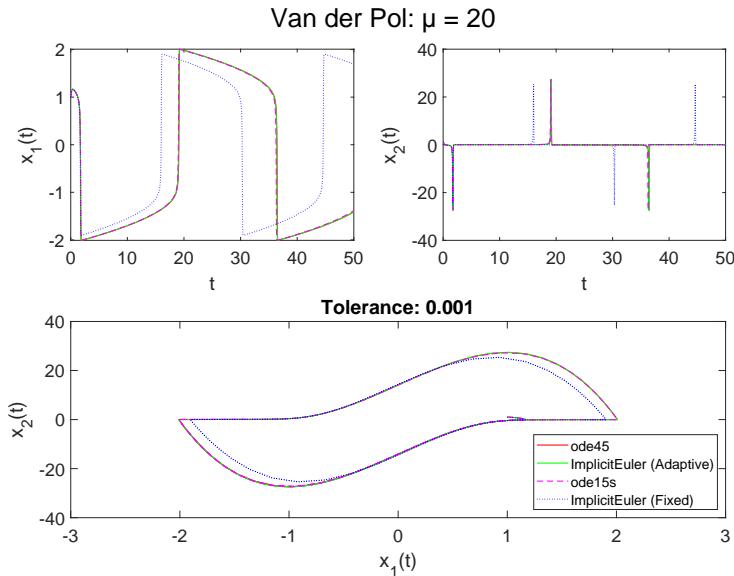
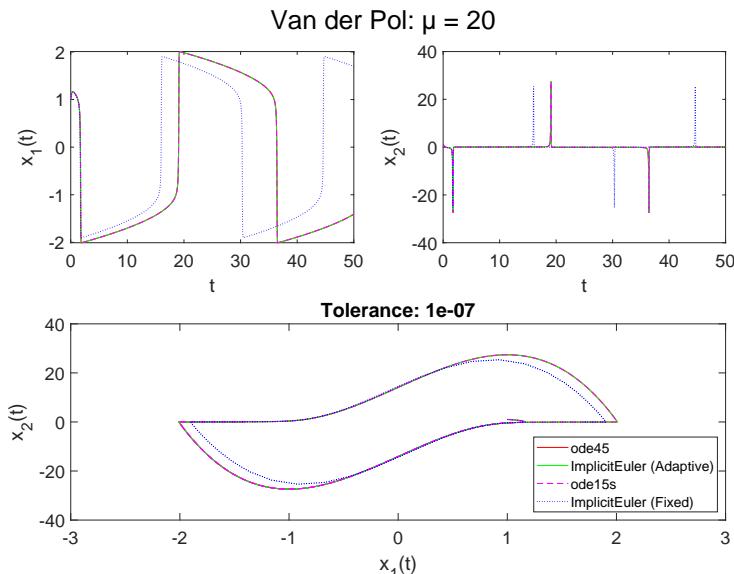


Figure 3.4: Comparison of explicit Euler with fixed and adaptive step size, *ode45* and *ode15s* on the Van der Pol problem, $\mu = 3$.



((a)) .



((b))

Figure 3.5: Comparison of implicit Euler with fixed and adaptive step size, `ode45` and `ode15s` on the Van der Pol problem, $\mu = 20$.

CHAPTER 4

Solvers for SDEs

In this chapter a stochastic differential equation (SDE) is considered in the form:

$$dx(t) = f(t, x(t), p_f)dt + g(t, x(t), p_g)d\omega(t) \quad d(\omega) \sim N_{iid}(0, Idt) \quad (4.1)$$

where $x \in \mathbb{R}^{n_x}$ and ω is a stochastic variable with dimension n_ω . p_f and p_g are parameters for $f : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \mapsto \mathbb{R}^{n_x}$ and $g : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \mapsto \mathbb{R}^{n_x \times n_\omega}$. In the preceding sections a multivariate standard Wiener process is realized. The SDE methods, Euler's explicit-explicit and implicit-explicit are then derived and tested on an SDE version of the Van der Pol problem.

4.1 Multivariate Standard Wiener Process

A Wiener process is the mathematical model for Brownian motion. The standard Wiener process, over $[0, T]$ is a random variable $\omega(t)$ that depends continuously on $t \in [0, T]$ and satisfies the following 3 conditions:

1. $\omega(0) = 0$ (with probability 1)
2. $0 \leq s < t \leq T : [\omega(t) - \omega(s)] \sim N(0, t - s)$
3. For $0 \leq s < t < u < v \leq T$, the increments $[\omega(t) - \omega(s)]$ and $[\omega(v) - \omega(u)]$ are independent.

A small increment of this process, $d\omega(t)$, is called white noise and Brownian motion is integrated white noise:

$$\omega(t) = \int_0^t d\omega(s). \quad (4.2)$$

In this section a multivariate standard Wiener function is realized. The properties of independence and stationarity of increments make the Wiener process easy to simulate. Although, the main challenge is that it is a continuous-time stochastic process yet computer simulations are discrete [7]. Thus, for computational purposes the discretized Wiener process is considered, where $\omega(t)$ is specified at discrete t values. This can be done for (4.2) above by first setting:

$$\Delta t = \frac{T}{N}, \quad (4.3)$$

for a positive integer N and then letting ω_k denote $\omega(t_k)$ with $t_k = k\Delta t$. Now, Condition 1 above says $\omega_0 = 0$ with probability 0 and given conditions 2 and 3 above the following is obtained:

$$\omega_k = \omega_{k-1} + d\omega_k, \quad j = 1, 2, \dots, N, \quad (4.4)$$

where each $d\omega_k$ is an independent and identically distributed (iid) random variable of the form $\sqrt{\Delta t}N(0, 1)$. The above derivations are based on section 2 on "Brownian Motion" in [8].

A function was implemented in Matlab that can realize a multivariate standard Wiener process. This was done by using the built-in function *rng* and including a parameter nW to allow the resulting motion to be multivariate, see the implemented code in Appendix A.1.3.

4.2 The Explicit-Explicit Method

The explicit-explicit method is often called the Euler-Maryuama method. In the same way as the Wiener process in Section 4.1 was discretized the explicit-explicit method can be obtained by discretizing the SDE in (4.1). This can be done by first letting $\nabla t = t/N$ for some positive integer N and then denoting the numerical approximations $x(t_k)$ as x_k with $t_k = k\nabla t$. The method then takes the form:

$$x_{k+1} = x_k + f(x_k)\Delta t_k + g(x_k)\Delta \omega_k, \quad \Delta \omega \sim N_{iid}(0, I\Delta t_k). \quad (4.5)$$

To better understand where this term comes from consider the integral form of (4.1):

$$x_k = x_{k+1} + \int_{k+1}^k f(x(t))dt + \int_{k+1}^k g(x(t))d\omega(t), \quad (4.6)$$

and notice that the terms on the right-hand side of (4.6) are approximated by each of the corresponding three terms on the same side in (4.5) [8]. The procedure for this method is listed in Algorithm 6 below and the Matlab code can be found in Appendix A.1.3.

Algorithm 6: Explicit-Explicit method for SDEs.

Data: Timespan T , x_0 , ω , function f , function g

Result: x

- 1 Set $x_1 = x_0$
- 2 Obtain number of time steps N from T
- 3 **for** i from 1 to $N-1$ **do**
- 4 $f_i = f(T_i, x_i)$ *// Evaluate function*
- 5 $g_i = g(T_i, x_i)$
- 6 $\nabla t = T_{i+1} - T_i$
- 7 $\nabla \omega = \omega_{i+1} - \omega_i$
- 8 $\psi = x_i + g \nabla \omega$
- 9 Update x
- 10 $x_{i+1} = \psi + f_i \nabla t$
- 11 **end**

4.3 The Implicit-Explicit Method

The implicit-explicit method can be derived in a similar way as for the explicit-explicit method, taking the form:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + f(\mathbf{x}_{k+1})\Delta t_k + g(\mathbf{x}_k)\Delta \omega_k, \quad \Delta \omega \sim N_{iid}(0, I\Delta t_k), \quad (4.7)$$

The deterministic term of the implicit-explicit method is solved as in the implicit Euler method, with the residual equation taking the form:

$$R_k(x_{k+1}) = x_{k+1} - f(x_{k+1})\Delta t_k - x_k + g(x_k)\Delta \omega_k = 0. \quad (4.8)$$

This system of nonlinear equations can then be solved by Newton's method [5]. The full procedure for the implicit-explicit method can be seen in Algorithm 7 and the corresponding Newton approximation in Algorithm 8. The Matlab code with the implementation can be found in Appendix A.1.3.

Algorithm 7: Implicit-Explicit method for SDEs.

Data: Timespan T , x_0 , ω , function f , function g
Result: x

- 1 Obtain number of time steps N from T
- 2 Set $\epsilon = 10^{-8}$ // Tolerance for Newton step
- 3 Set $x_1 = x_0$
- 4 $f_1 = f(T_1, x_1)$ // Evaluate function
- 5 **for** i from 1 to $N-1$ **do**
- 6 $g_i = g(T_i, x_i)$ // Evaluate function
- 7 $\nabla t = T_{i+1} - T_i$
- 8 $\nabla \omega = \omega_{i+1} - \omega_i$
- 9 $\psi = x_i + g \nabla \omega$
- 10 $x_{init} = \psi + f_i \nabla t$
- 11 $x_{i+1} = SDENewtonSolver(f, t_{i+1}, \nabla t, \psi, x_{init}, \epsilon)$ // Obtain next step using Newton's method
- 12 **end**

Algorithm 8: Newton's method to obtain next step in the implicit-explicit method.

Procedure $SDENewtonSolver$

Data: Function F , t , ∇t , ψ , x_{init} , tolerance ϵ
Result: x

- 1 **Procedure** $SDENewtonSolver$
- 2 Set $x = x_{init}$
- 3 $[f, J] = F(t, x)$ // Evaluate function at current point
- 4 Set $R = x - f * \nabla t - \psi$ // Initialize residual term
- 5 **while** $\|R\|_\infty > \epsilon$ **do**
- 6 $\frac{\partial R}{\partial x} = I - J * h$
- 7 $\nabla x = R / \frac{\partial R}{\partial x}$
- 8 $x := x - \nabla x$
- 9 $[f, J] = F(t, x)$ // Evaluate function at current point
- 10 $R := x - fh - x_i$ // Update residual term
- 11 **end**

4.4 SDE Version of the Van der Pol Problem

The Van der Pol problem can be converted to an SDE by the addition of a diffusion term. The SDE version of the Van der Pol problem as a state independent and dependent diffusion is given by:

$$d\mathbf{x}_1(t) = \mathbf{x}_2(t)dt \quad (4.9)$$

$$d\mathbf{x}_2(t) = [\mu(1 - \mathbf{x}_1(t)^2)\mathbf{x}_2(t) - \mathbf{x}_1(t)]dt + \sigma d\omega(t) \quad (4.10)$$

$$d\mathbf{x}_2(t) = [\mu(1 - \mathbf{x}_1(t)^2)\mathbf{x}_2(t) - \mathbf{x}_1(t)]dt + \sigma(1 + \mathbf{x}_1(t)^2)d\omega(t) \quad (4.11)$$

In the preceding sections, the methods derived above are tested on the SDE version of the Van der Pol problem. This was done for a non-stiff system with $\mu = 3$ and a stiff version $\mu = 20$ and large versus small step sizes compared, $N = [1000, 10000]$. The parameter $\sigma = 0.5$ and the initial point was set to $x_0 = [0.5, 0.5]^T$. The number of realizations were chosen to be 5 for easier visualization. The driver for performing the tests can be found in Appendix A.2.4.

4.4.1 The SDE Methods on the Van Der Pol Problem

The results can be seen in the figures below. The black lines represent the drift term, corresponding to the ODE version of the Van der Pol, $\sigma = 0$. Figure 4.1 and Figure 4.3 show that the methods have good performance for the non-stiff system with both small and large steps. However, when it comes to the stiff system in Figure 4.2 and Figure 4.4, then when the step size gets too large the system becomes very unstable. Although, notice how much better the implicit-explicit method performs for the larger steps. To conclude, the implicit-explicit method is a better choice when it comes to the stiff system. This seems to be the case for implicit methods in general as the implicit Euler method derived in Chapter 3 also performed well for the stiff system.

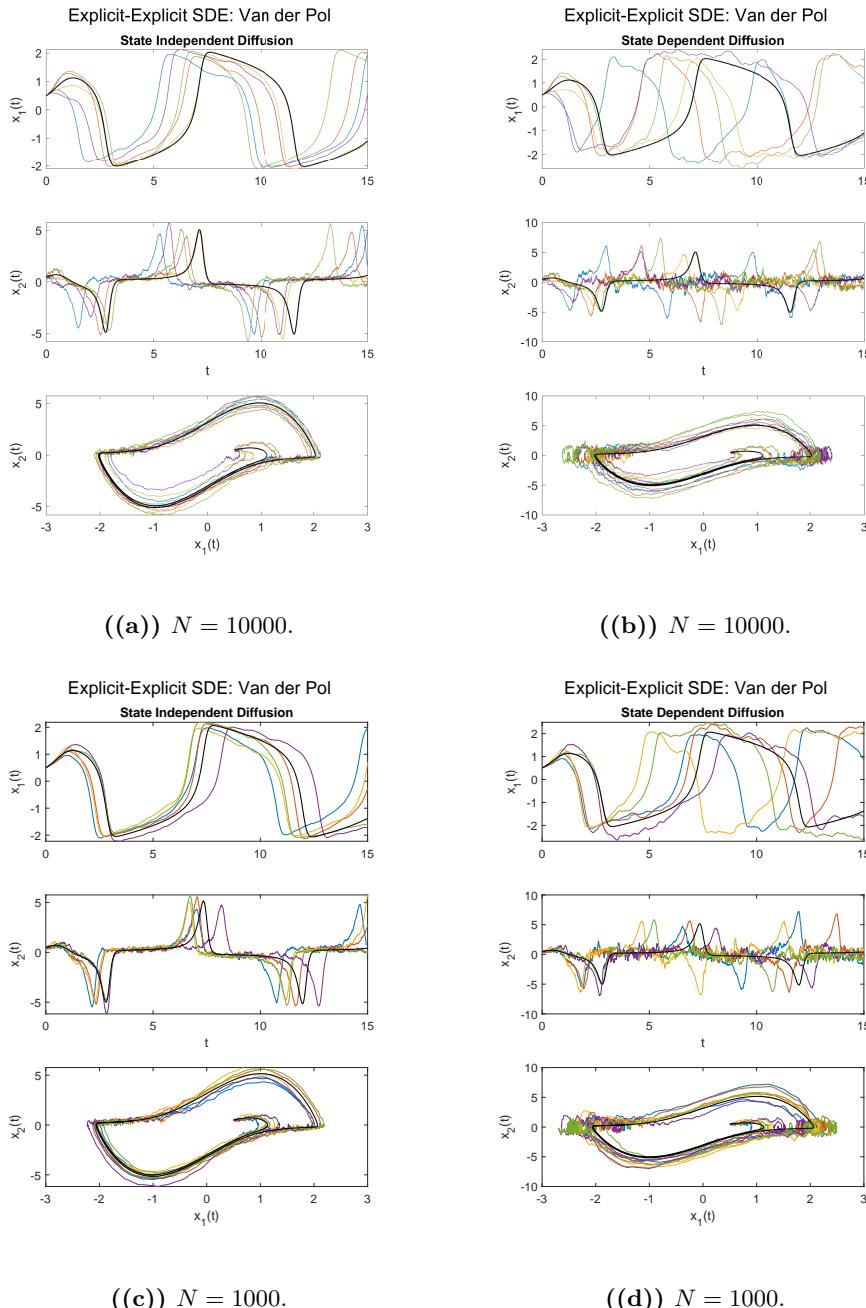


Figure 4.1: Explicit-Explicit Euler with fixed step size on an SDE version of the Van der Pol problem with $\mu = 3$ and $\sigma = 0.5$. The black lines show the drift term ($\sigma = 0$).

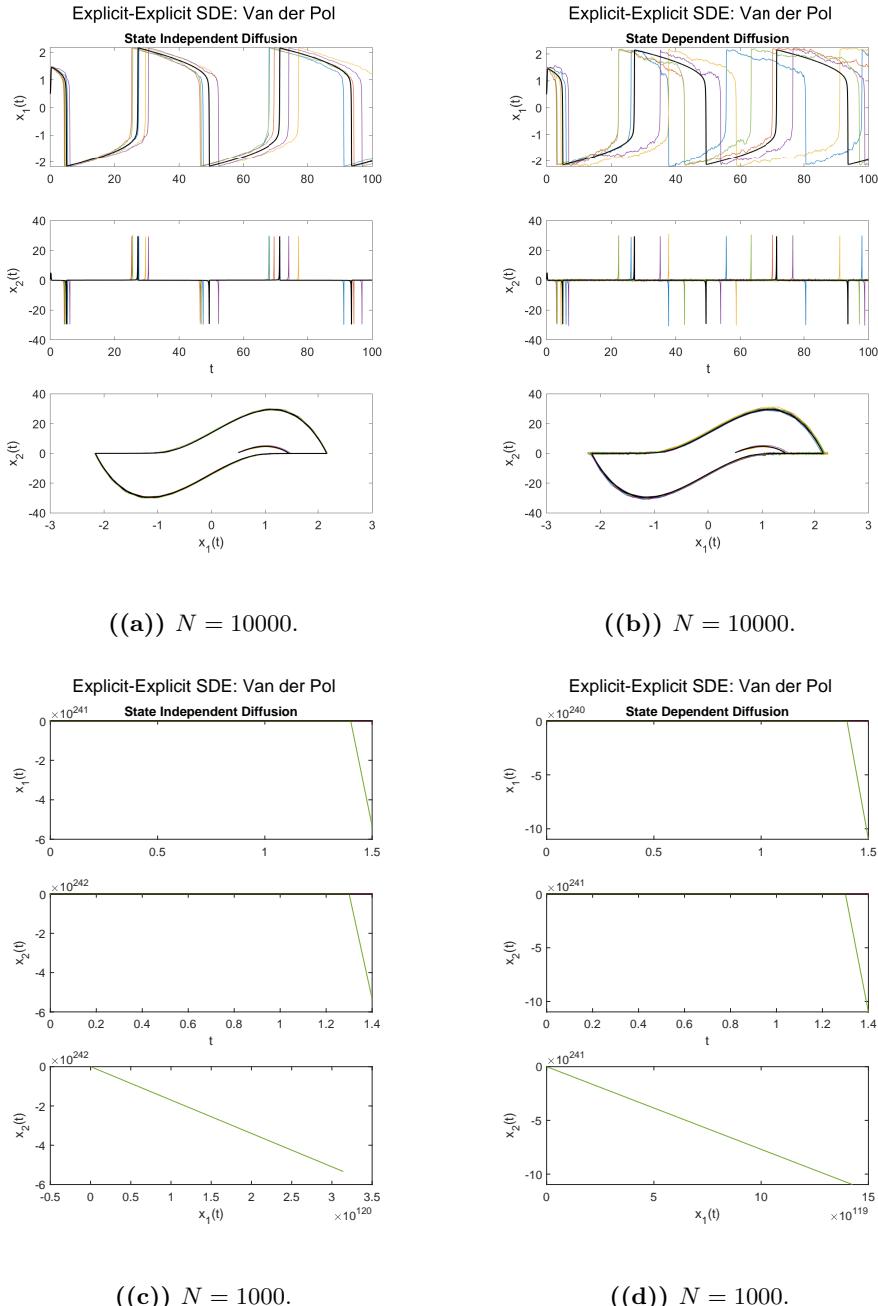


Figure 4.2: Explicit-Explicit Euler with fixed step size on an SDE version of the Van der Pol problem with $\mu = 20$ and $\sigma = 0.5$. The black lines show the drift term ($\sigma = 0$).

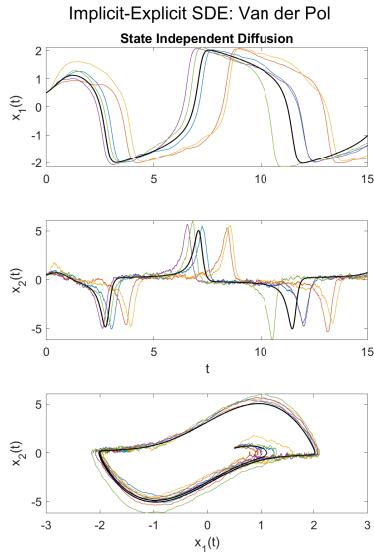
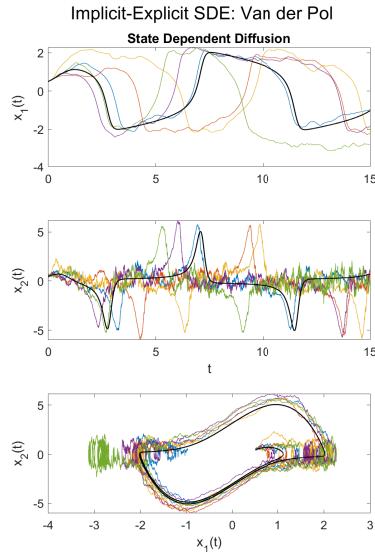
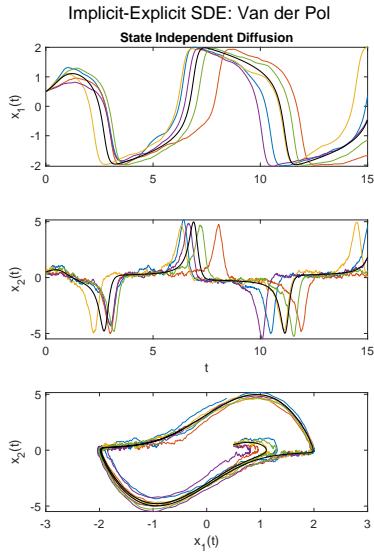
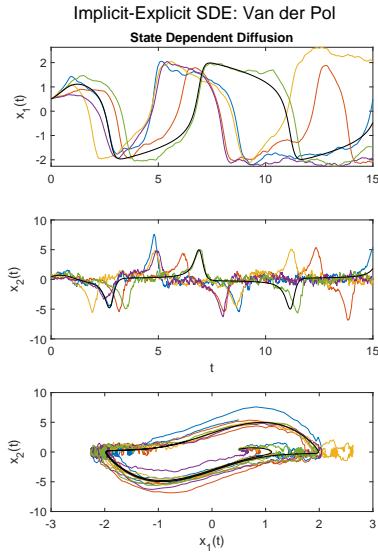
((a)) $N = 10000$.((b)) $N = 10000$.((c)) $N = 1000$.((d)) $N = 1000$.

Figure 4.3: Implicit-Explicit Euler with fixed step size on an SDE version of the Van der Pol problem with $\mu = 3$ and $\sigma = 0.5$. The black lines show the drift term ($\sigma = 0$).

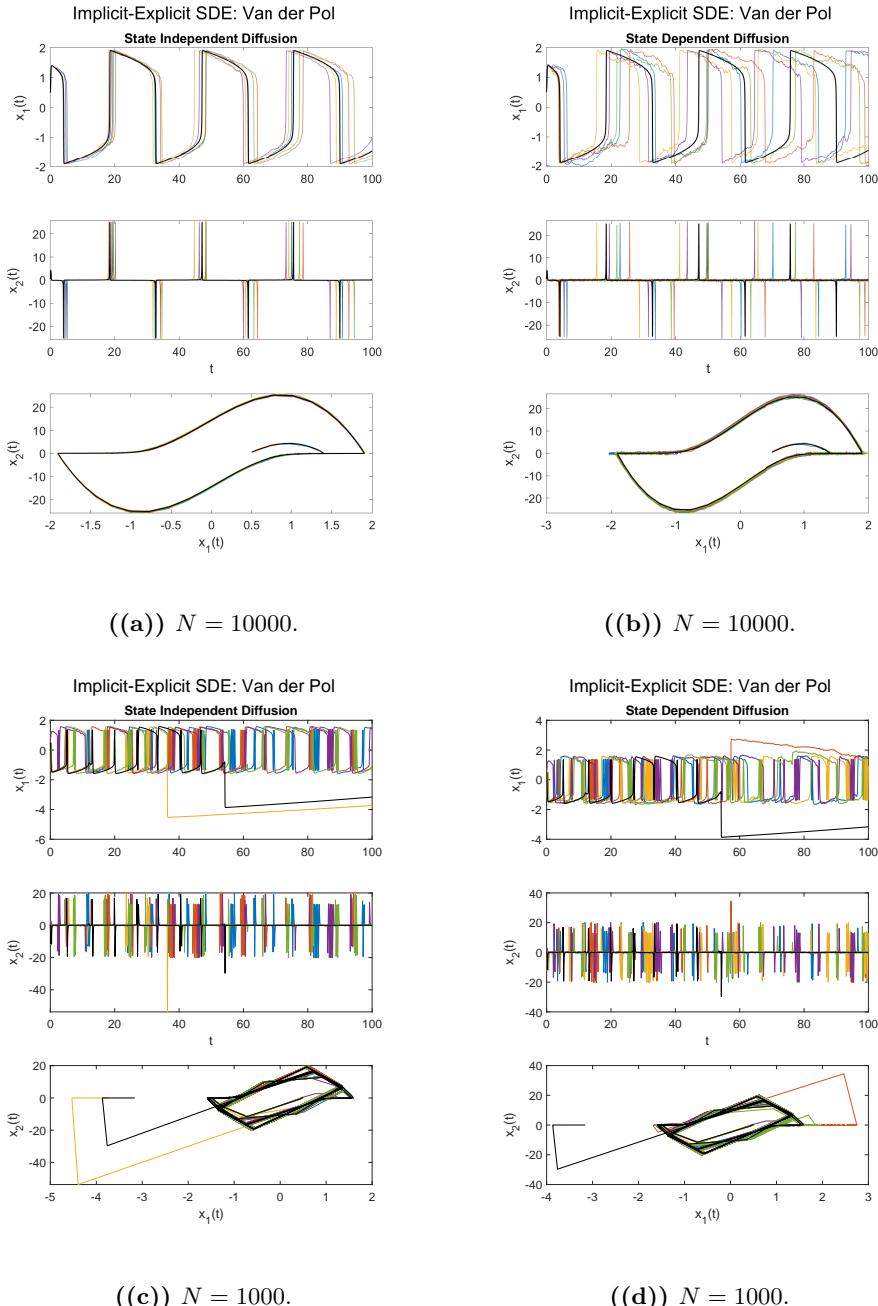


Figure 4.4: Implicit-Explicit Euler with fixed step size on an SDE version of the Van der Pol problem with $\mu = 20$ and $\sigma = 0.5$. The black lines show the drift term ($\sigma = 0$).

CHAPTER 5

Classical Runge-Kutta Method

In this chapter, the initial value problem is considered:

$$\dot{x}(t) = f(t, x(t), p), \quad x(t_0) = x_0, \quad (5.1)$$

where $x \in \mathbb{R}^{n_x}$ and $p \in \mathbb{R}^{n_p}$.

5.1 The Classical Runge-Kutta

In general, an s -stage Runge-Kutta method for the ODE system presented in (5.1) can be written in the form:

$$X_i = x_{n-1} + h \sum_{j=1}^s a_{ij} f(t_{n-1} + c_j h, X_j) \quad (5.2a)$$

$$x_n = x_{n-1} + h \sum_{i=1}^s b_i f(t_{n-1} + c_i h, X_i) \quad (5.2b)$$

where a_{ij} are the internal coefficients and b_i for $i = 1, \dots, s$ are the weights. The stage values, X_i , are intermediate approximations to the solution x at times $t_{n-1} + c_i h$, and c_i is always chosen as [1]:

$$c_i = \sum_{j=1}^s a_{ij}, \quad i = 1, \dots, s \quad (5.3)$$

Now, letting $T_i = t_n + c_i h$ the method can be reformulated into a simpler form:

$$X_i = x_{n-1} + h \sum_{j=1}^s a_{ij} f(T_j, X_j) \quad (5.4)$$

$$x_n = x_{n-1} + h \sum_{i=1}^s b_i f(X_i) \quad (5.5)$$

The above coefficients are chosen such that the error terms cancel out and x_n is becomes more accurate [1]. In this chapter, the classical Runge-Kutta method is considered. It is a fourth-order Runge-Kutta method with $s = 4$ stages which means it evaluates f four times at each step. According to Lecture 8 in [5] it takes the form:

$$X_1 = x_n \quad (5.6a)$$

$$X_2 = x_n + \frac{h}{2}f(X_1) \quad (5.6b)$$

$$X_3 = x_n + \frac{h}{2}f(X_2) \quad (5.6c)$$

$$X_4 = x_n + hf(X_3) \quad (5.6d)$$

$$x_{n+1} = x_n + h\left(\frac{1}{6}f(X_1) + \frac{1}{3}f(X_2) + \frac{1}{3}f(X_3) + \frac{1}{6}f(X_4)\right). \quad (5.6e)$$

The formulas in (5.6) can be represented in what is called a *Butcher tableau* which for the fourth order Runge-Kutta method can be seen in Table 5.1.

Table 5.1: The Butcher tableau for the classical Runge-Kutta method.

0	0	0	0	0
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
$\frac{1}{2}$	0	$\frac{1}{2}$	0	0
1	0	0	1	0
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$

5.2 Classical Runge Kutta with Fixed Step Size

To get a better overview of how the classical Runge-Kutta works for approximating solutions to (5.1), see Algorithm 9 which presents the algorithm for the method with a fixed step size. The Matlab code for this method can be found in Appendix A.1.4.

Algorithm 9: Classical Runge-Kutta with fixed time step.

Data: A, b , and c from Butcher tableau, h , x_0 , objective function f , t_0, t_f

Result: t, x

- 1 Set $N = \frac{t_f - t_0}{h}$
- 2 Set $x = x_0$
- 3 **for** $n=1\dots N$ **do**
- 4 **Stage 1**
- 5 Set $T_1 = t_0$
- 6 Set $X_1 = x_0$
- 7 $F_1 = f(T_1, X_1)$ // Evaluate function at given point
- 8 **Stage 2-4**
- 9 **for** $i=2,\dots,4$ **do**
- 10 $X_i = x + h \sum_{j=1}^{i-1} a_{ij} F_j$ // a_{ij} from the A matrix
- 11 $T_i = t + hc_i$
- 12 $F_i = f(T_i, X_i)$
- 13 **end**
- 14 **Compute next step**
- 15 $t := t + h$
- 16 $x := x + h \sum_{i=1}^4 b_i F_i$ // b_i from the b vector
- 17 **end**

5.2.1 Classical Runge-Kutta with Adaptive Step Size

As for the Euler methods in Chapter 2 and Chapter 3, the classical Runge-Kutta can also be modified to use adaptive steps and error estimation by step doubling. The same procedure applies as in the aforementioned chapters. The full step is the same as explained in (5.5) but to further elaborate, the double step is computed using the following equations [9]:

$$\begin{aligned}\hat{T}_i^{n+\frac{1}{2}} &= t_n + c_i \frac{h}{2} \\ \hat{X}_i^{n+\frac{1}{2}} &= x_n + \frac{h}{2} \sum_{j=1}^{i-1} a_{ij} f(\hat{T}_j^{n+\frac{1}{2}}, \hat{X}_j^{n+\frac{1}{2}}) \\ \hat{x}_{n+\frac{1}{2}} &= x_n + \frac{h}{2} \sum_{j=1}^{i-1} b_i f(\hat{T}_j^{n+\frac{1}{2}}, \hat{X}_j^{n+\frac{1}{2}})\end{aligned}$$

$$\begin{aligned}\hat{T}_i^{n+1} &= t_n + c_i \frac{h}{2} \\ \hat{X}_i^{n+1} &= x_n + \frac{h}{2} \sum_{j=1}^{i-1} a_{ij} f(\hat{T}_j^{n+1}, \hat{X}_j^{n+1}) \\ \hat{x}_{n+1} &= \hat{x}_{n+1}^{n+\frac{1}{2}} + \frac{h}{2} \sum_{j=1}^{i-1} b_j f(\hat{T}_j^{n+1}, \hat{X}_j^{n+1})\end{aligned}$$

and finally the local truncation error is estimated as in (2.3). The full procedure is explained in Algorithm 11 along with the corresponding step function Algorithm 10. The Matlab code for this method is listed in Appendix A.1.4.

Algorithm 10: One step of the classical Runge-Kutta.

```

1 Procedure: ClassicalRungeKuttaStep
  Data:  $A$ ,  $b$ ,  $c$  from Butcher tableau,  $x$ ,  $t$ , objective function  $f$ , step size  $h$ 
  Result:  $t, x$ 
2 Stage 1
3 Set  $T_1 = t$ 
4 Set  $X_1 = x$ 
5  $F_1 = f(T_1, X_1)$  // Evaluate function at given point
6 Stage 2-4
7 for  $i=2,\dots,4$  do
8    $X_i = x + h \sum_{j=1}^{i-1} a_{ij} F_j$  //  $a_{ij}$  from the  $A$  matrix
9    $T_i = t + hc_i$ 
10   $F_i = f(T_i, X_i)$ 
11 end
12 Compute next step
13  $t := t + h$ 
14  $x := x + h \sum_{i=1}^4 b_i F_i$  //  $b_i$  from the  $b$  vector

```

Algorithm 11: Classical Runge-Kutta with adaptive time step and error estimation.

Data: A, b, c from Butcher tableau, initial step size h_0 , x_0 , objective function F , t_0, t_f , abstol and reltol

Result: t, x

```

1 Set  $\epsilon = 0.8$                                      // Tolerance
2 Set  $\mathcal{F}_{min} = 0.1$                          // Maximum decrease factor
3 Set  $\mathcal{F}_{max} = 5.0$                          // Maximum increase factor
4 Set  $h = h_0$ 
5 Set  $t = t_0$ 
6 Set  $x = x_0$ 
7 while  $t < t_f$  do
8   if  $t + h > t_f$  then
9     |  $h = t_f - t$ 
10  end
11   $f=F(t,x)$                                 // Evaluate function at current point
12  while Step is NOT accepted do
13     $x_{next} = ClassicalRungeKuttaStep(A,b,c,t,x,F,h)$  // Take full step
14     $h_{temp} = \frac{1}{2}h$ 
15     $[t_{temp}, x_{temp}] = ClassicalRungeKuttaStep(A,b,c,t,x,F,h_{temp})$  // Take half step
16     $f_{temp} = F(t_{temp}, x_{temp})$                       // Evaluate function
17     $[\hat{t}, \hat{x}] = ClassicalRungeKuttaStep(A,b,c,t_{temp},x_{temp},F,h_{temp})$ 
18     $e = x_{next} - \hat{x}$ 
19     $r = \max\left\{\frac{|e|}{\max\{abstol, |\hat{x}| \cdot reltol\}}\right\}$ 
20    if  $r \leq 1.0$  then
21      | Step is accepted - update point
22      |  $t := t + h$ 
23      |  $x := \hat{x}$ 
24    end
25    Update step size using asymptotic step size controller
26     $h := \max\{\mathcal{F}_{min}, \min\{\sqrt{\epsilon/r}, \mathcal{F}_{max}\}\}h$ 
27  end
28 end

```

5.3 Testing the Classical Runge-Kutta on the Van der Pol Problem

In this section, the classical Runge-Kutta methods derived above are tested on the Van der Pol problem given in (2.6) with initial point $x_0 = [1, 1]^T$. This was done for both stiff and non-stiff versions of the system and for a time span of $t = [0, 50]$. The driver for performing the tests can be found in Appendix A.2.5.

5.3.1 Classical Runge-Kutta with Fixed Step Size

The classical Runge-Kutta with fixed step size was tested on the Van der Pol problem with a step size of $N = 3000$ and $N = 10000$ steps, with the same setup as in previous chapters. The results can be seen in Figure 5.1. The phase plot in Figure 5.1(b) is quite smooth with some sharp edges at times. Compared to the explicit Euler method (see Figure 2.1(b)), the Runge-Kutta manages to represent the system more accurately, despite the small step size. On the other hand, the implicit Euler method manages to obtain smoother fluctuations in the solution for the stiff system, see Figure 3.1(b). As for the non-stiff system, there is no apparent difference between using smaller or larger steps in the results and the fluctuations are nice and smooth.

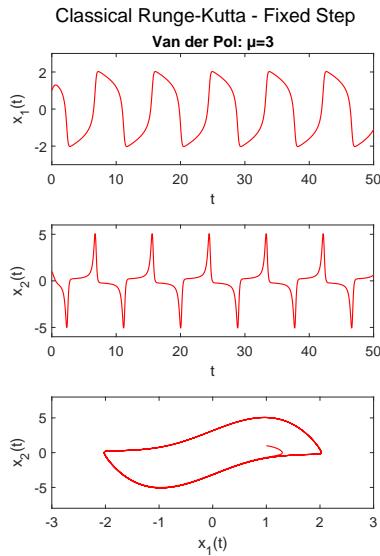
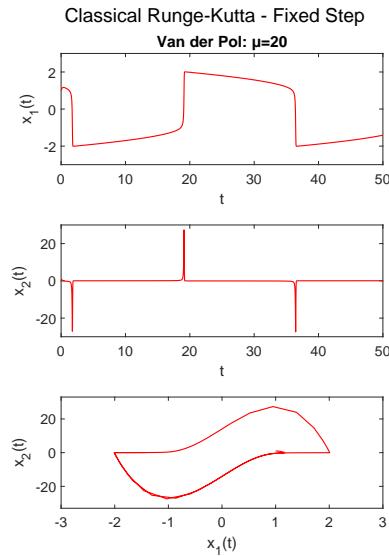
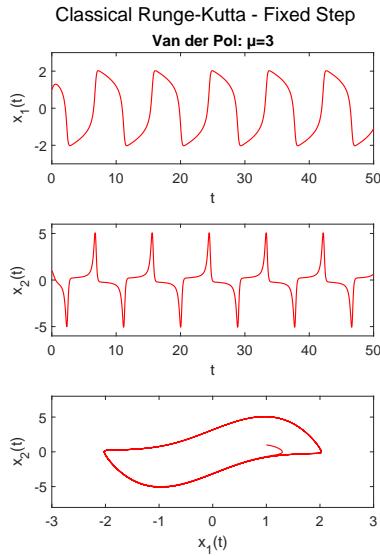
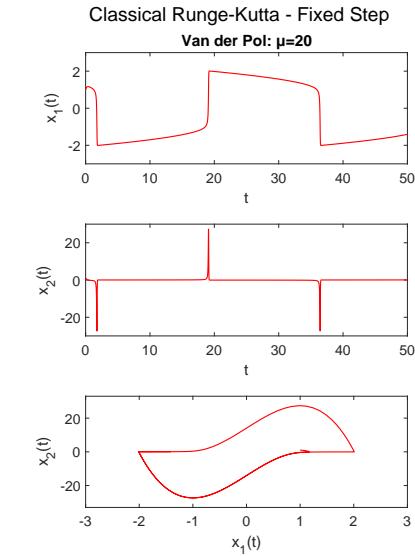
((a)) $h = 0.0167$.((b)) $h = 0.0167$.((c)) $h = 0.005$.((d)) $h = 0.005$.

Figure 5.1: Classical Runge-Kutta with fixed step size, h , on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.

5.3.2 Classical Runge-Kutta with Adaptive Step Size

The classical Runge-Kutta with adaptive step size was tested on the Van der Pol problem with initial step size of $h_0 = 0.001$ as in previous chapters. This was done with absolute and relative tolerance of 10^{-5} . The results can be seen in Figure 5.2 and the corresponding error estimator and step sizes in Figure 5.3. Figure 5.2(b) shows that using adaptive steps instead of fixed has increased the accuracy of the solution such that the phase plot is smoother. On the other hand, the non-stiff system is presented in the same way as for the fixed steps with no noticeable difference in the outcome. As for the error estimation it can be seen that the behaviour is more erratic than for the explicit and implicit Euler methods, see Figure 2.3 and Figure 3.3. In general, the classical Runge-Kutta is unsuitable for stiff systems as it has a small region of absolute stability [10]. This could be the reason for the erratic behaviour. To conclude, the classical Runge-Kutta performs well when the problem is non-stiff and for stiff systems an implicit method would be preferable.

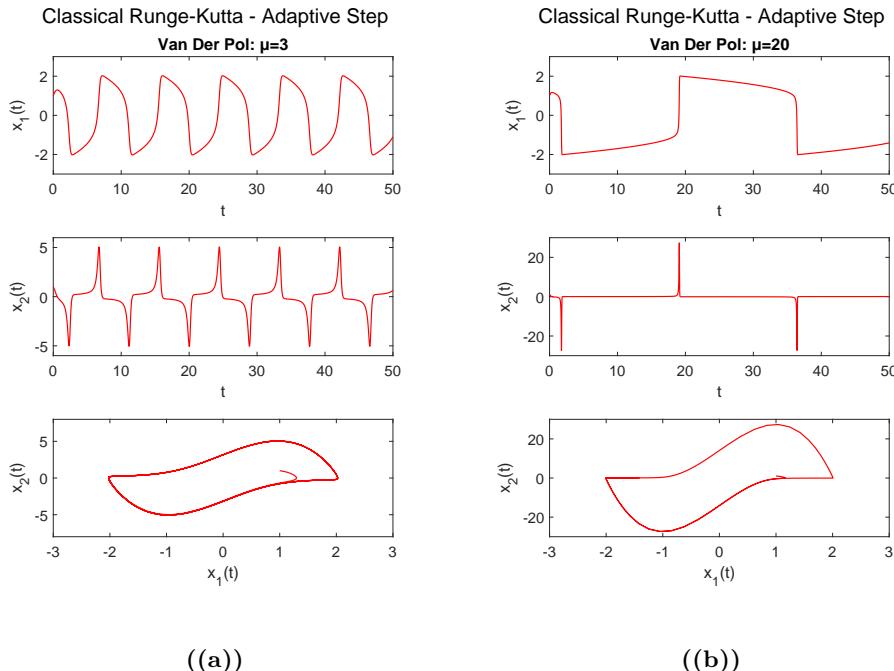
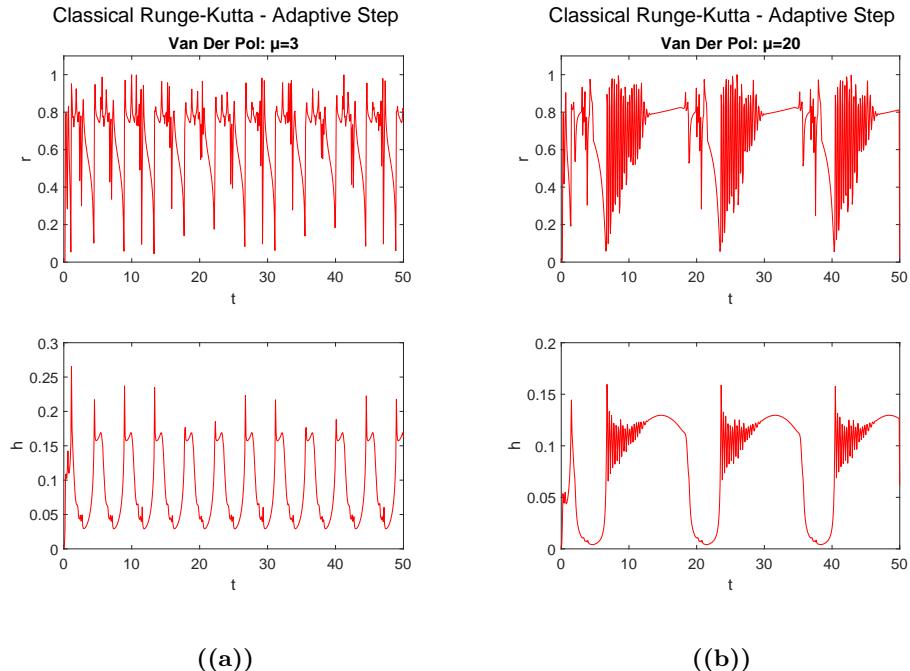


Figure 5.2: Classical Runge-Kutta with adaptive step size, $h_0 = 0.001$, on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.



((a))

((b))

Figure 5.3: Error estimator, r , and step sizes, h , using the classical Runge-Kutta with adaptive step size on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.

5.3.3 Comparing with Matlab's ODE Solvers

As the method worked fast for low tolerances it was decided to test it against Matlab's ODE solvers for even lower tolerances than in previous chapters, and the absolute and relative tolerances tested were: 10^{-3} and 10^{-12} . Table 5.3 confirms the above statement that this method works quite fast for low tolerances. Overall, the method requires more function evaluations than the ODE solvers, see Table 5.2.

Table 5.2: Comparison of classical Runge-Kutta with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-3} .

	Runge-Kutta		<i>ode45</i>		<i>ode15s</i>	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
N. Function	3562	6364	1489	3751	750	1393
N. Steps	332	596	248	625	675	363
N. Accepted	242	404	194	583	559	287
N. Rejected	90	192	54	42	116	76

Table 5.3: Comparison of classical Runge-Kutta with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-12} .

	Runge-Kutta		<i>ode45</i>		<i>ode15s</i>	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
N. Function	153600	144680	60157	57733	13520	29124
N. Steps	13970	13155	10026	9622	16116	7339
N. Accepted	13901	13132	10016	9619	15916	7196
N. Rejected	69	23	10	3	200	143

The visualizations in Figure 5.4 and Figure 5.5 show that the classical Runge-Kutta manages with both fixed and adaptive steps to maintain the same accuracy as the ODE solvers for low tolerances. For higher tolerances, according to the phase plots the method seems to shift slightly for both the non-stiff and stiff Van der Pol system.

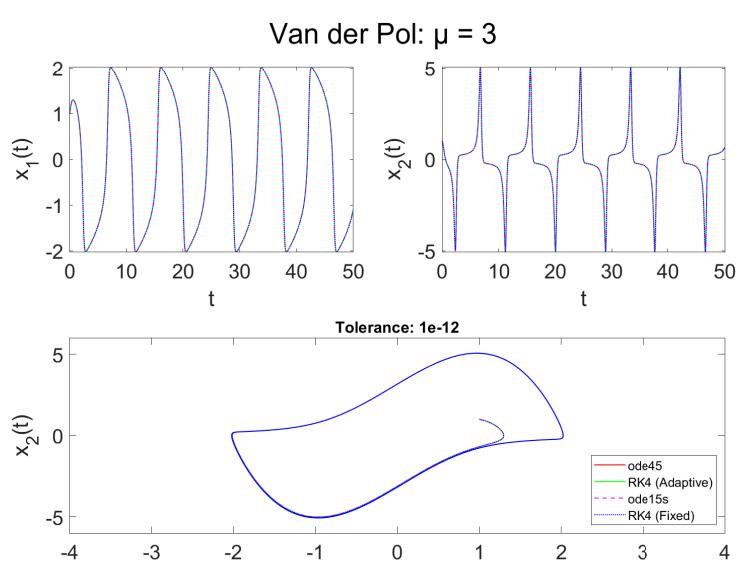
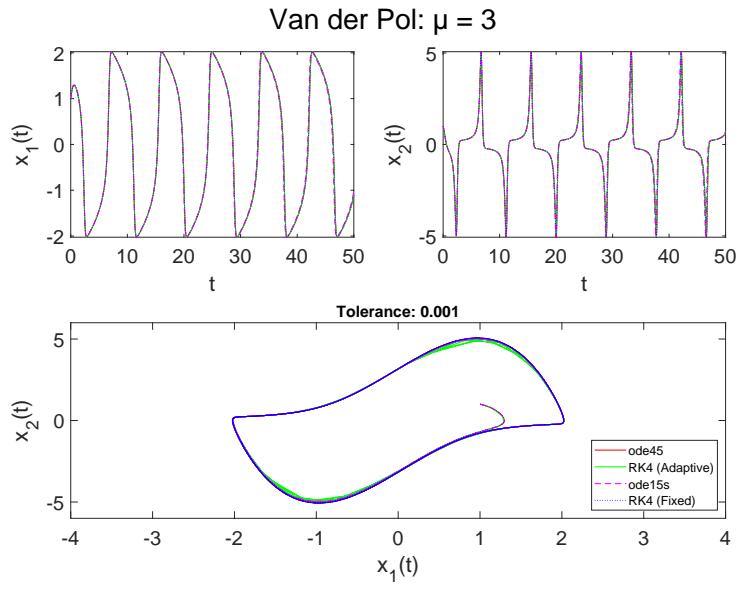


Figure 5.4: Comparison of classical Runge-Kutta with fixed and adaptive step size, *ode45* and *ode15s* on the Van der Pol problem, $\mu = 3$.

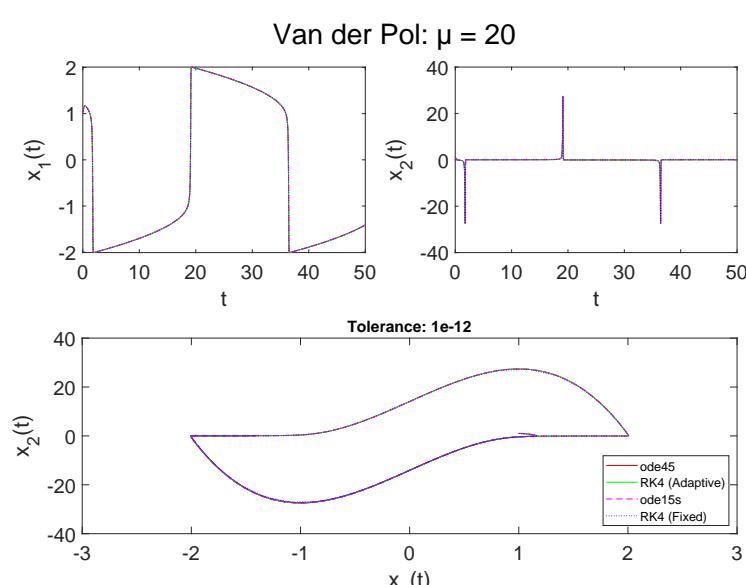
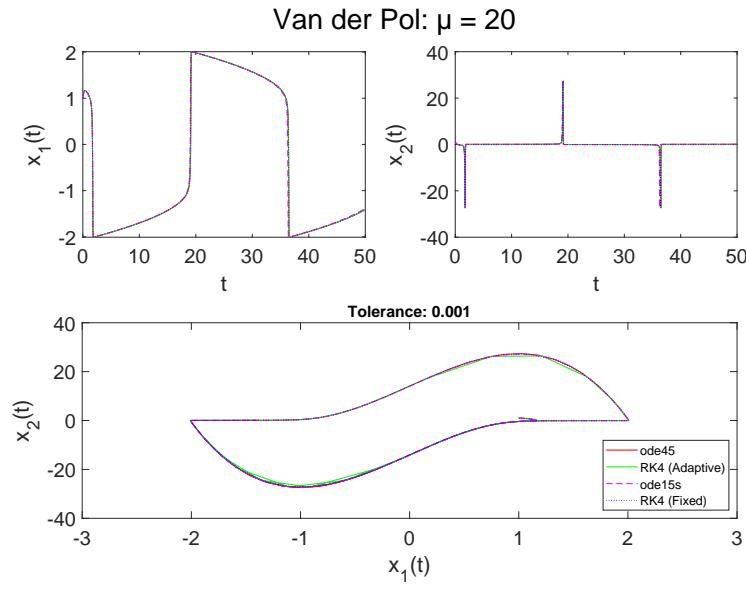


Figure 5.5: Comparison of classical Runge-Kutta with fixed and adaptive step size, *ode45* and *ode15s* on the Van der Pol problem, $\mu = 20$.

CHAPTER 6

Dormand-Prince 5(4)

In this chapter the following initial value problem is considered:

$$\dot{x} = f(t, x(t), p), \quad x(t_0) = x_0, \quad (6.1)$$

where $x \in \mathbb{R}^n$ and $p \in \mathbb{R}^{n_p}$.

6.1 DOPRI54

The Dormand-Prince method, or DOPRI method, is a member of the Runge-Kutta family of ODE solvers. It has seven stages and uses six function evaluations to calculate fourth- and fifth-order calculations, hence the name *DOPRI54*. It is also the default method in Matlab's *ode45* solver. The Butcher tableau can be seen in Table 6.1. The general procedure defined in (5.2) in Chapter 5 applies to this method where a and b are obtained from the Butcher tableau.

Table 6.1: Butcher tableau for the DOPRI54 method. The lowest row represents the error d .

0	0	0	0	0	0	0	0
1/5	1/5	0	0	0	0	0	0
3/10	3/10	9/40	0	0	0	0	0
4/5	44/45	-56/15	32/9	0	0	0	0
8/9	19372/6561	-25360/2187	64448/6561	-212/729	0	0	0
1	9017/3168	-355/33	46732/5247	49/176	-5103/18656	0	0
1	35/384	0	500/1113	125/192	-2187/6784	11/84	0
	35/384	0	500/1113	125/192	-2187/6784	11/84	0
	5179/57600	0	7571/16695	393/640	-92097/339200	187/2100	1/40
	71/57600	0	-71/16695	-71/1920	-17253/339200	22/525	-1/40

6.1.1 DOPRI54 with Adaptive Step Size

The DOPRI54 can be modified to include an adaptive step size. It is an embedded method and therefore uses embedded error estimation. The concept is similar to step doubling. For a p order method the local error can be estimated using a Runge-Kutta method of one order higher, $p + 1$. As for step doubling the error estimate is defined as [9]:

$$e = \hat{x}_{n+1} - x_{n+1}. \quad (6.2)$$

The only difference is that now there is no need to calculate \hat{x}_{n+1} since it only depends on the coefficient d which components can be found in the last row in Table 6.1. That is:

$$e = \hat{x}_{n+1} - x_{n+1} \quad (6.3)$$

$$= (x_n + h \sum_{j=1}^r b_j f(T_j, X_j)) - (x_n + h \sum_{j=1}^r \hat{b}_j f(T_j, X_j)) \quad (6.4)$$

$$= h \sum_{j=1}^r (b_j - \hat{b}_j) f(T_j, X_j) \quad (6.5)$$

$$= h \sum_{j=1}^r d_j f(T_j, X_j) \quad (6.6)$$

This makes it easier to implement the method as the error estimation is built into the Runge-Kutta step. The asymptotic step controller is computed depending on the order of the method and for DOPRI54 it is defined as:

$$h_{n+1} = \left(\frac{\epsilon}{r_{n+1}} \right)^{1/5} h_n. \quad (6.7)$$

The full procedure for the DOPRI54 method with embedded error estimation and adaptive step size is explained in Algorithm 12. The method was implemented in Matlab and the code is listed in Appendix A.1.5. The code itself is structured such that the step with the embedded error estimation is in a separate function and it outputs the approximation of the next step along with the error. Stats are included in the output such that number of function evaluations, step size and error estimation can be assessed.

In the following sections, the implemented DOPRI54 solver is tested on a variety of problems in order to assess its performance. The tolerance used was 10^{-5} . The driver for performing the tests can be found in Appendix A.2.6.

Algorithm 12: DOPRI54 with adaptive time step.

Data: A, b, c from Butcher tableau, initial step size h_0 , x_0 , objective function F , t_0, t_f , abstol and reltol

Result: t, x, e

```

1 Set  $s = 7$                                 // Number of stages in method
2 Set  $\epsilon = 0.8$                          // Tolerance
3 Set  $\mathcal{F}_{min} = 0.1$                   // Maximum decrease factor
4 Set  $\mathcal{F}_{max} = 5.0$                   // Maximum increase factor
5 Set  $h = h_0$ 
6 Set  $t = t_0$ 
7 Set  $x = x_0$ 
8 while  $t < t_f$  do
9   | if  $t + h > t_f$  then
10    |   |  $h = t_f - t$ 
11   | end
12   |  $f = F(t, x)$                       // Evaluate function at current point
13   | while Step is NOT accepted do
14     |   | Take step of size  $h$  with embedded error estimation
15     |   | Set  $T_1 = t$ 
16     |   | Set  $X_1 = x$ 
17     |   | Set  $f_1 = f$ 
18     |   | for  $i = 2, \dots, s$  do
19       |   |   |  $X_i = x + h \sum_{j=1}^{i-1} a_{ij} f_j$           //  $a_{ij}$  from the A matrix
20       |   |   |  $T_i = t + hc_i$ 
21       |   |   |  $f_i = F(T_i, X_i)$ 
22     |   | end
23     |   | Approximate next step and compute error
24     |   |  $e := h \sum_{j=1}^s d_j f_j$ 
25     |   |  $\hat{t} := t + c_7 h$ 
26     |   |  $\hat{x} := X_7$ 
27     |   | Error estimation
28     |   |  $r = \max\left\{\frac{|e|}{\max\{abstol, |\hat{x}| reltol\}}\right\}$ 
29     |   | if  $r \leq 1.0$  then
30       |   |   | Step is accepted - update point
31       |   |   |  $t := \hat{t} + h$ 
32       |   |   |  $x := \hat{x}$ 
33     |   | end
34     |   | Update step size using asymptotic step size controller
35     |   |  $h := \max\{\mathcal{F}_{min}, \min\left\{\left(\frac{\epsilon}{r_{n+1}}\right)^{1/5}, \mathcal{F}_{max}\right\}\}h$ 
36   | end
37 end

```

6.1.2 The Test Equation

In this section, the test equation described in Chapter 1 is used to assess the implemented DOPRI54 method. The initial step size used was $h_0 = 0.01$ and the solution was computed for a time span of $t = [0, 10]$. Figure 6.1 shows the DOPRI54 solution to the test equation versus the analytical solution. As can be seen the method manages to approximate the solution accurately.

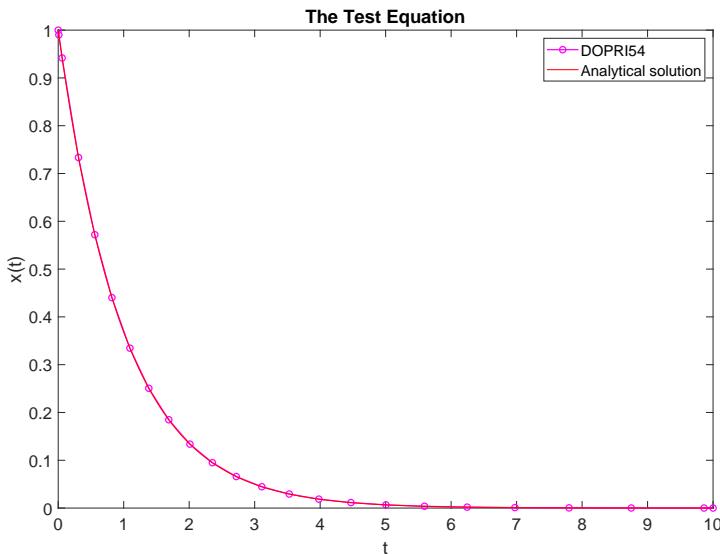


Figure 6.1: DOPRI54 solution to the test equation vs. the analytical solution.

As the analytical solution is known the order and stability of the method can be analyzed. To assess the order of the method, the exact local error was computed along with the one obtained using embedded error estimation and step doubling. Figure 6.2 shows the results. Step doubling manages to represent the exact local error with much precision which shows this method is more accurate. However, the embedded error estimator is less computationally expensive as it requires n computations while the step doubling requires approximately $3n$ computations. The figure shows the slope of the curve to be approximately 6 for the exact local error and step doubling. The local error is thus observed to be of complexity $\mathcal{O}(h^6)$ which indicates the method is of order 5. On the other hand the curve for the embedded error grows approximately by h^5 which indicates that the method is of order 4. The DOPRI54 method uses both fourth and fifth order calculations, which supports these findings.

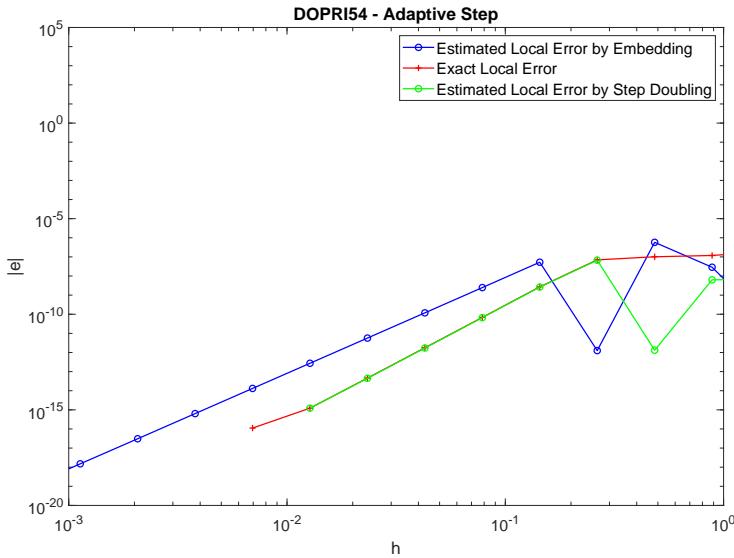


Figure 6.2: Log-log plot of the exact vs estimated local error of the DOPRI54 method.

The stability region of the DOPRI54 can be seen in Figure 6.3. The region resembles the one for the classical Runge-Kutta in Section 1.2. As was described in Section 1.2 the method is only A-stable if the entire left half plane is contained within the stability region but the figure shows that this is not true for DOPRI54. It is therefore not A-stable. This condition needs to be satisfied in order to have L-stability and thus the DOPRI54 method is also not L-stable. This is evident from the figure as According to Lecture 10 on "ESDIRK" methods [5], this is true in general for explicit Runge-Kutta methods.

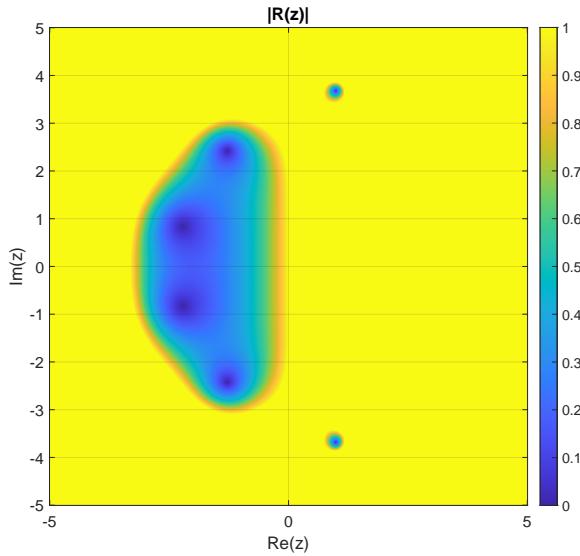


Figure 6.3: The stability region of the DOPRI54 method.

6.1.3 The Van der Pol Problem

The DOPRI54 solver was tested on the Van der Pol problem in (2.6). This was done for an initial step size of $h_0 = 0.001$ and for a stiff $\mu = 3$, and non-stiff system $\mu = 20$. The initial point was set to $x_0 = [1, 1]^T$ and the tolerance 10^{-5} . The results can be seen in Figure 6.4. The method manages to approximate the system quite efficiently with rapid fluctuations in the non-stiff system. This is similar to the results obtained using the classical Runge-Kutta in Chapter 5. This seems reasonable as the method is just one order higher. The error estimator and step sizes shown in Figure 6.5 also have erratic behaviour as in the results for the aforementioned method.

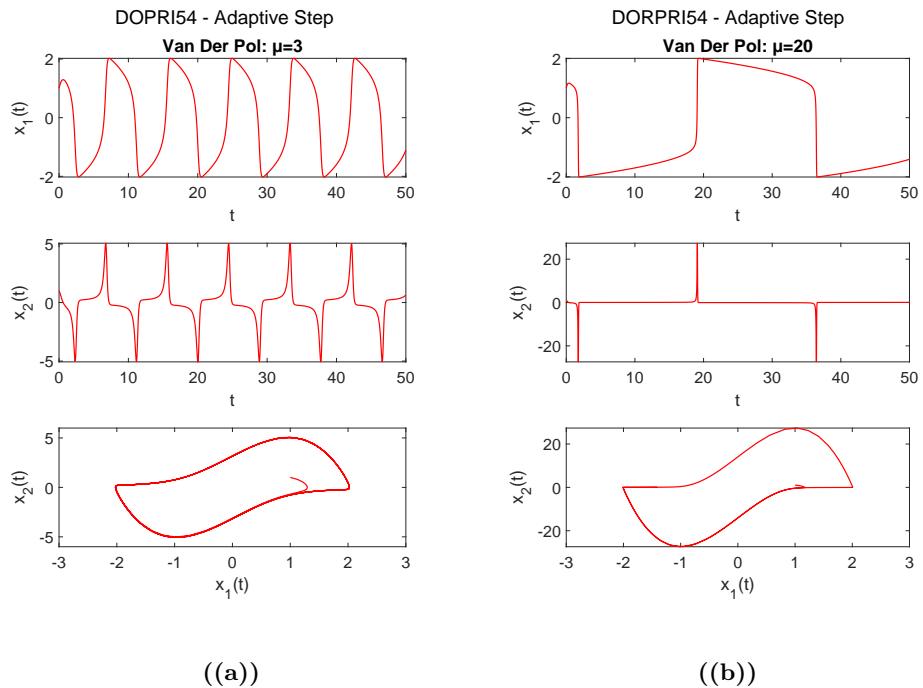


Figure 6.4: DOPRI54 with adaptive step size on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.

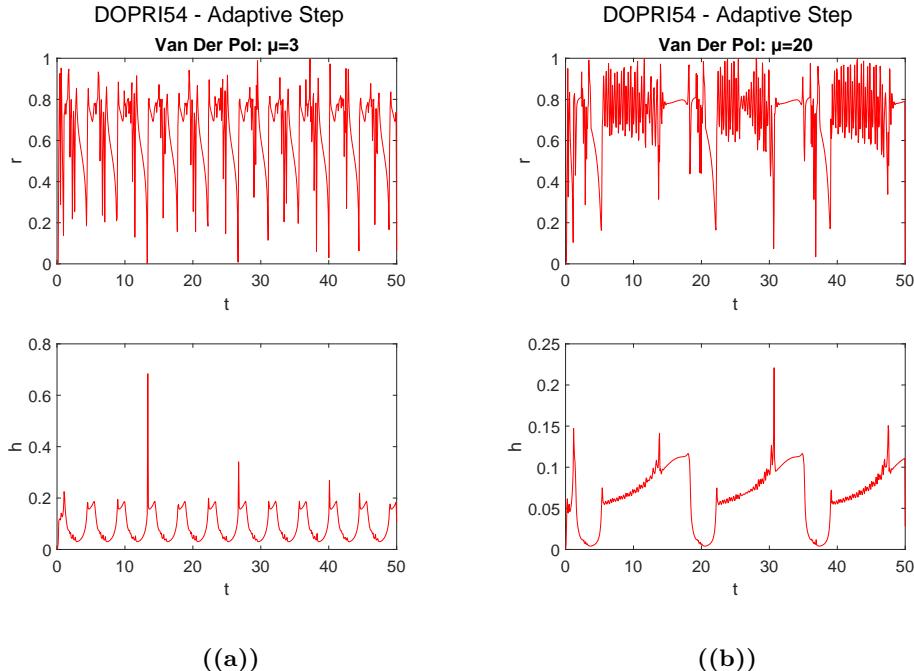


Figure 6.5: Error estimator, r , and step sizes, h , using the DOPRI54 with adaptive step size on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.

6.1.4 Adiabatic CSTR

In this section, the DOPRI54 method is tested on an adiabatic continuous stirred tank reactor (CSTR) with an exothermic reaction. The reaction consists of sodium thiosulfate and hydrogen peroxide in aqueous solutions. First, consider the 3D version of the system:

$$\dot{C}_A = \frac{F}{V}(C_{A,in} - C_A) + R_A(C_A, C_B, T), \quad (6.8a)$$

$$\dot{C}_B = \frac{F}{V}(C_{B,in} - C_B) + R_B(C_A, C_B, T), \quad (6.8b)$$

$$\dot{T} = \frac{F}{V}(T_{in} - T) + R_T(C_A, C_B, T), \quad (6.8c)$$

where T represents temperature, F is the flow rate, C_A and C_B represent the concentrations of substances A and B , and R represents the production rate. Now, C_A and C_B may be approximated by:

$$C_A = C_{A,in} = \frac{1}{\beta}(T_{in} - T) \quad (6.9a)$$

$$C_B = C_{B,in} = \frac{2}{\beta}(T_{in} - T) \quad (6.9b)$$

$$(6.9c)$$

The evolution of the system can then be described by the 1D differential equation:

$$\dot{T} = \frac{F}{V}(T_{in} - T) + R_T(C_A(T), C_B(T), T), \quad (6.10a)$$

For more detail on the CSTR system, the parameters used and it's derivations, see paper in [11]. The Matlab code for the 1D and 3D versions of the CSTR can be found in Appendix A.1.5. The results can be seen in Figure 6.6. The solutions are almost identical, with the 1D solution lagging slightly behing after 10 minutes but manages to catch up quickly. Interestingly, a decrease in the flow rate causes a spike in the temperature.

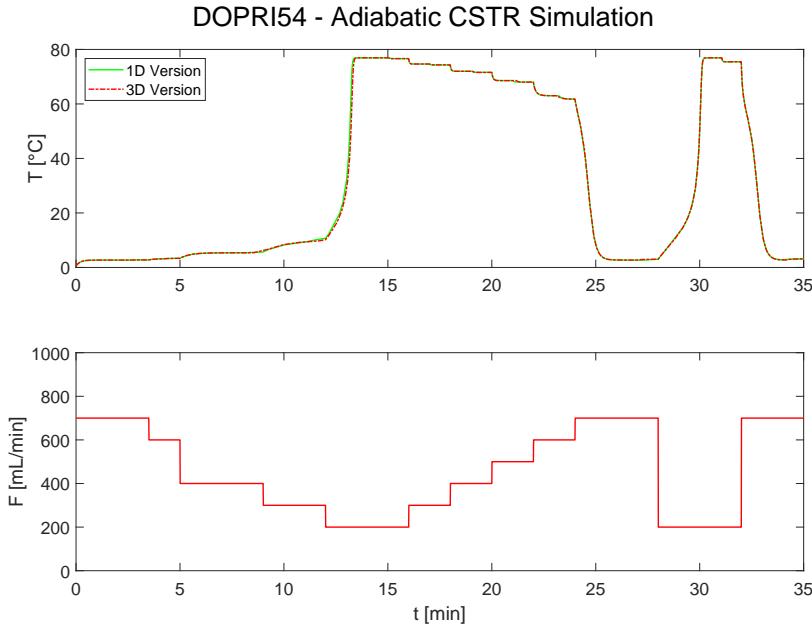


Figure 6.6: Simulation of adiabiatic CSTR using the DOPRI54 method, 1D vs. 3D version.

6.1.5 Comparing with Matlab's ODE Solvers

The implementation of the DOPRI54 method was compared with Matlab's ODE solvers, *ode45* and *ode15s* to assess it's performance. Table 6.2 and Table 6.3 show that the method is almost identical in behaviour to Matlab's *ode45*. DOPRI54 is the default solver in *ode45*, which explains the results. This shows that the method is working as it should. Despite the low tolerance the method is working efficiently, with similar number of function evaluations as the Euler methods with much higher tolerance.

Table 6.2: Comparison of DOPRI54 with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-3} .

	DOPRI54		ode45		ode15s	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
N. Function	1693	4399	1489	3751	1393	750
N. Steps	252	646	248	625	675	363
N. Accepted	181	583	194	583	559	287
N. Rejected	71	63	54	42	116	76

Table 6.3: Comparison of DOPRI54 with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-12} .

	DOPRI54		ode45		ode15s	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
N. Function	59494	56780	60157	57733	29124	13520
N. Steps	8517	8117	10026	9622	16116	7339
N. Accepted	8399	8078	10016	9619	15916	7196
N. Rejected	118	39	10	3	200	143

Figure 6.7 and Figure 6.8 show the comparison of the approximations from the three methods. The DOPRI54 method seems to catch up with the fluctuations of the other methods but it's phase portrait has sharp edges at times for the high tolerance. From Figure 6.8 it seems the method performs better for the stiff system at higher tolerances. For a low tolerance the three methods have identical solutions, as was anticipated.

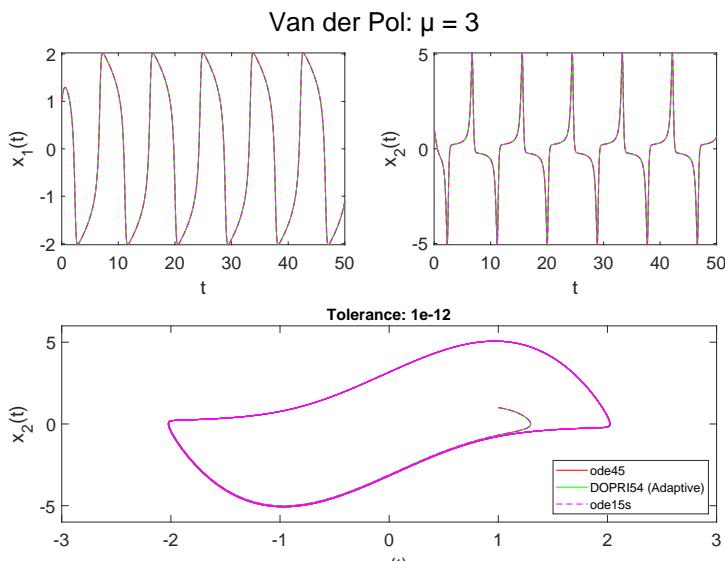
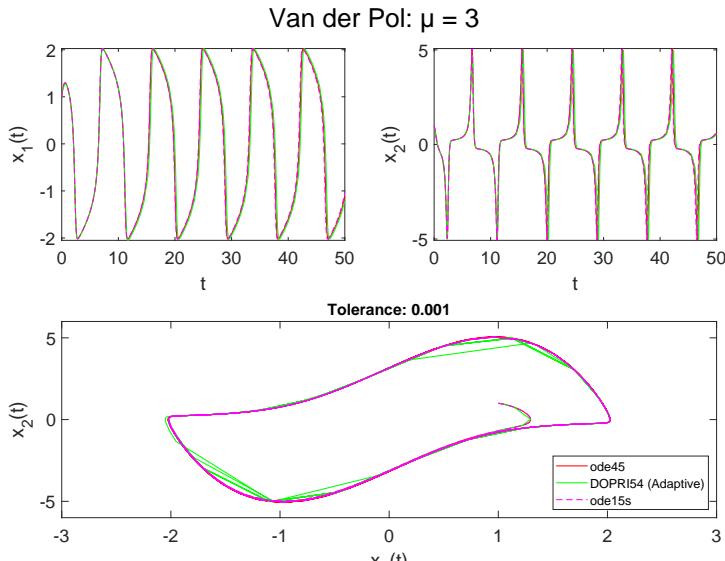


Figure 6.7: Comparison of DOPRI54, *ode45* and *ode15s* on the Van der Pol problem, $\mu = 3$.

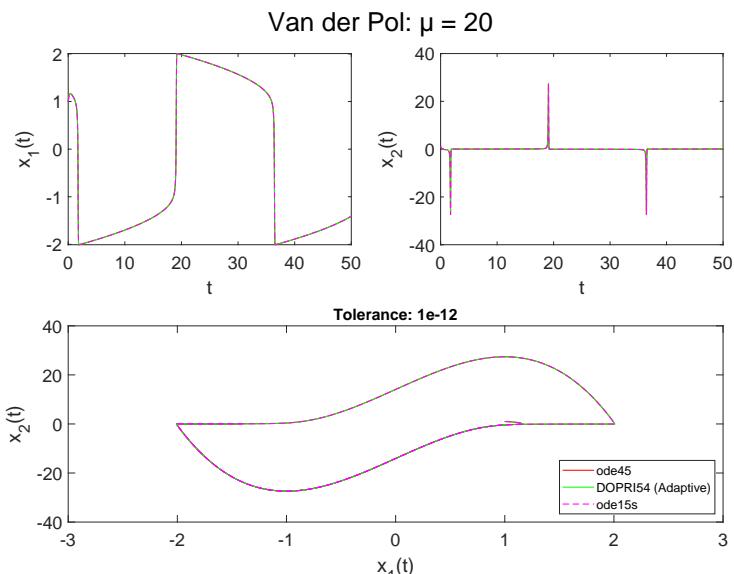
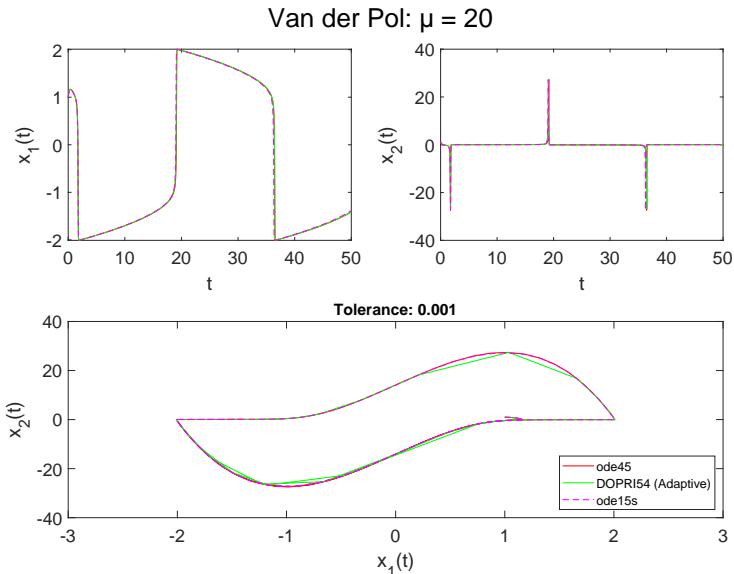


Figure 6.8: Comparison of DOPRI54 with fixed and adaptive step size, *ode45* and *ode15s* on the Van der Pol problem, $\mu = 20$.

CHAPTER 7

ESDIRK23

ESDIRK stands for explicit singly diagonal implicit Runge–Kutta. It is a series of explicit methods with implicit inner stages. As the diagonal elements of these methods are identical, the iteration matrix can be reused, which saves computational cost. In the following sections, the ESDIRK23 method will be derived and its performance and stability assessed. The driver for this chapter can be found in Appendix A.2.7.

7.1 ESDIRK23

ESDIRK23 is a three-stage implicit Runge-Kutta method of order two and three. The general butcher tableau for this method is represented in Table 7.1.

Table 7.1: General Butcher tableau for the ESDIRK23 method.

0	0	0	0
c_2	a_{21}	γ	0
1	b_1	b_2	γ
	b_1	b_2	γ
	\hat{b}_1	\hat{b}_2	\hat{b}_3
	d_1	d_2	d_3

The parameters can be derived using the order and consistency conditions of the method. The following derivations are based on the paper "A family of ESDIRK integration methods" in [12]. The stability function for the ESDIRK23 method is defined as:

$$R(z) = \frac{(1 + b_1 + b_2 - \gamma)z + (a_{21}b_2 - b_1\gamma)z^2}{(1 - \gamma z)^2}. \quad (7.1)$$

The aim is to construct a method that satisfies the conditions for L-stability. According to Section 1.2, in order to have L-stability then $\lim_{\infty} R(z) = 0$, and thus the following requirement needs to hold:

$$a_{21}b_2 - b_1\gamma = 0. \quad (7.2)$$

That is the numerator order in (7.2) must be less than the denominator order. The consistency requirement for the scheme of the ESDIRK23 method is defined as:

$$c_2 = a_{21} + \gamma, \quad (7.3a)$$

$$1 = b_1 + b_2 + \gamma. \quad (7.3b)$$

The order conditions are:

$$1st\ order : b_1 + b_2 + \gamma = 1, \quad (7.4a)$$

$$2nd\ order : b_2 c_2 + \gamma = \frac{1}{2}. \quad (7.4b)$$

Now, stage 2 of the method is required to be of order 2 and thus the following conditions need to be satisfied:

$$a_{21} + \gamma = c_2, \quad (7.5a)$$

$$\gamma c_2 = \frac{1}{2} c_2, \quad (7.5b)$$

which are equivalent to $a_{21} = \gamma$ and $c_2 = 2\gamma$. This means consistency is now automatically provided by the order conditions in (7.4) and (7.5). Notice that the first order condition is identical to the consistency requirement in (7.3b). The conditions above therefore yield 5 nonlinear equations with 5 unknown variables b_1, b_2, c_2, a_{21} , and γ . The requirement of $0 < c_2 < 1$ then yields the unique solution $\gamma = \frac{2-\sqrt{2}}{2}$. Now, an embedded method of order 3 may be determined as the solution to the following conditions:

$$1st\ order : \hat{b}_1 + \hat{b}_3 + \hat{b}_3 = 1, \quad (7.6a)$$

$$2nd\ order : \hat{b}_2 c_2 + \hat{b}_3 = \frac{1}{2}, \quad (7.6b)$$

$$3rd\ order : \hat{b}_2 c_2^2 + \hat{b}_3 = \frac{1}{3}, \quad (7.6c)$$

$$\hat{b}_2(c_2\gamma) + \hat{b}_3(c_2 b_2 + \gamma) = \frac{1}{6}. \quad (7.6d)$$

As the two equations for the 3rd order condition in (7.6) above are linearly dependent the conditions constitute 3 linear equations with 3 unknown variables \hat{b}_1, \hat{b}_2 , and \hat{b}_3 . The parameters in Table 7.1 can now be solved for and are summarized in Table 7.2.

Table 7.2: The Butcher tableau for the ESDIRK23 with $\gamma = \frac{2-\sqrt{2}}{2}$.

0	0	0	0
2γ	γ	γ	0
1	$\frac{1-\gamma}{2}$	$\frac{1-\gamma}{2}$	γ
	$\frac{1-\gamma}{2}$	$\frac{1-\gamma}{2}$	γ
	$\frac{6\gamma-1}{12\gamma}$	$\frac{1}{12\gamma(1-2\gamma)}$	$\frac{1-3\gamma}{3(1-2\gamma)}$
	$\frac{1-6\gamma^2}{12\gamma}$	$\frac{6(1-2\gamma)(1-\gamma)-1}{12\gamma(1-2\gamma)}$	$\frac{6\gamma(1-\gamma)}{3(1-2\gamma)}$

7.1.1 Stability

The stability region of the ESDIRK23 method was visualized to assess it's stability properties. The results can be seen in Figure 7.1. The stability region is contained within the yellow region of the figure. Evidently, the method is A-stable as the region is contained within the entire left-half plane of the figure. Additionally, computing the highest value for $Re(z) < 0$ yielded 0.99 which fits with the stability requirements explained in Chapter 1. Solving for the stability function in (7.2) yielded $\lim_{z \rightarrow -\infty} |\mathcal{R}(z)| = 0$ and the two conditions for L-stability are therefore satisfied. To summaries, the method is both A-stable and L-stable.

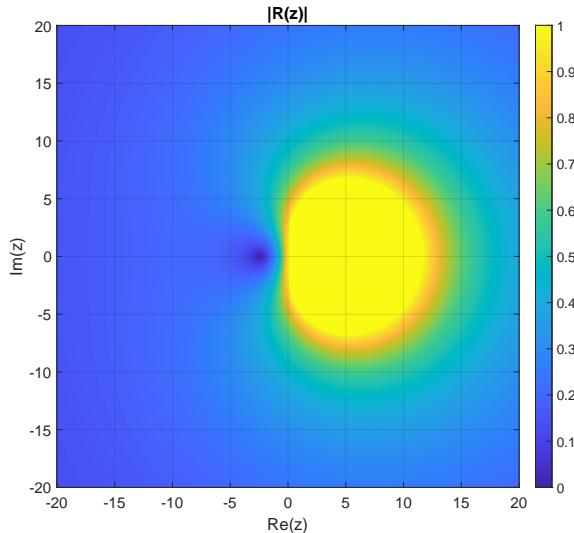


Figure 7.1: The stability region of the ESDIRK23 method.

7.1.2 ESDIRK23 with Variable Step Size

In this section, the ESDIRK23 method is modified to include an adaptive step size. The following derivations are based on Lecture 10 on "Runge-Kutta Methods" [5]. The general scheme for the ESDIRK23 can be written out as:

$$T_1 = t_n \quad X_1 = x_n \quad (7.7)$$

$$T_2 = t_n + 2\gamma h \quad X_2 = x_n + ha_{21}f(T_1, X_1) + h\gamma f(T_2, X_2) \quad (7.8)$$

$$\begin{aligned} T_3 &= t_n + h \\ X_3 &= x_n + ha_{31}f(X_1, T_1) + ha_{32}f(T_2, X_2) \\ &\quad + h\gamma f(T_3, X_3) \end{aligned} \quad (7.9)$$

The next state and the error are then estimated using the scheme above. The second and the third stages are implicit as indicated in Section 7.1.2. To solve these stages, an inexact Newton solver is used which approximates the Jacobian by:

$$M \approx I - h\gamma J(t_n, x_n), \quad (7.10)$$

where J is the Jacobian of function f . This makes the method less expensive as now LU factorization can be used to compute the inverse of M . It approximates the next iterate by minimizing the residual function:

$$R_i = X_i - h\gamma f(T_i, X_i) - \psi_i, \quad i = 2, 3, \quad (7.11)$$

where:

$$\psi_2 = x_n + ha_{21}f(T_1, X_1), \quad (7.12)$$

$$\psi_3 = x_n + ha_{31}f(T_1, X_1) + ha_{32}f(T_2, X_2). \quad (7.13)$$

An initial guess is required for X_2 which is found by one step of Euler's method:

$$X_2 = x_n + c_2 h f(t_n, x_n) = x_n + c_2 h f_1. \quad (7.14)$$

To ensure convergence in as few iterations as possible, the convergence rate is monitored by the parameter α , defined as:

$$\alpha = \max_n \frac{\|R_i X^{n+1}\|}{\|R_i X^n\|}. \quad (7.15)$$

In case of divergence, $\alpha \geq 1$, the iteration sequence is terminated. Otherwise the procedure is performed until $\|R_i\| < \epsilon$. After the Newton method converges, the next iterate is calculated by:

$$x_{n+1} = X_3, \quad (7.16)$$

$$t_{n+1} = T_3. \quad (7.17)$$

An asymptotic error controller is used to keep track of the error. The error estimate is:

$$\hat{e}_{n+1} = h \sum_{j=1}^s d_j f_j. \quad (7.18)$$

The measure of the error estimate is calculated as:

$$r = \max_i \left\{ \frac{|e_i|}{abstol_i + |(x_n)|_i reltol_i} \right\}. \quad (7.19)$$

In case the step is accepted, $r < 1$, the method uses a predictive error controller to update the step size:

$$h_{n+1} = \left(\frac{h_n}{r_{n+1}} \right) \left(\frac{\epsilon}{r_{n+1}} \right)^{1/(p+1)} \left(\frac{r}{r_{n+1}} \right)^{1/(p+1)} h_n, \quad (7.20)$$

where p is the order of the method $p = 2$ and $\epsilon = 0.8$. In case of a rejected step, the step size is updated by the asymptotic error controller described in previous chapters:

$$h_{n+1} = \left(\frac{\epsilon}{r_{n+1}} \right)^{1/(p+1)}. \quad (7.21)$$

One step of the ESDIRK23 method is summarized in Algorithm 13. The code is listed in Appendix A.1.6, it is derived from the one provided by the teacher [5].

Algorithm 13: One step of the ESDIRK23 method using inexact Newton's method.

```

1 Procedure: ESDIRK23Step
  Data:  $A, c, d$  from Butcher tableau,  $t_n, x_n, h, \gamma, \epsilon$ , objective function  $F$ 
  Result:  $t_{n+1}, x_{n+1}, e_{n+1}$ 
2 Compute internal stages
3 Set  $\alpha = 0$  Set  $X_1 = x_n$ 
4 Set  $f_1 = F(t_n, x_n)$ 
5 Set  $J = \frac{\partial f}{\partial x}(t_n, x_n)$ 
6 Set  $M = I - h\gamma J$ 
7 Set  $[L, U] = lu(M)$  // LU factorize M
8 for  $i = 2 : 3$  do
  9  $\psi_i = x_n + h \sum_{j=1}^{i-1} a_{ij} f_j$ 
  10  $T_i = t_n + c_i h$ 
  11  $X_i = x_n + (c_i h) f_1$ 
  12  $R_i = X_i - h\gamma f_i - \psi_i$ 
  13 while  $\|R_i\| > \epsilon$  AND  $\alpha < 1$  do
    14  $\Delta X_i = -U^{-1}(L^{-1} R_i)$  // Solve using LU-factorization
    15  $X_i = X_i + \Delta X_i$ 
    16  $f_i = f(T_i, X_i)$ 
    17  $R_i = X_i - h\gamma f_i - \psi_i$ 
    18  $\alpha = \max\{\alpha, \frac{R_{i-1}}{R_i}\}$  // Monitor convergence rate
  19 end
20 end
21 Compute next step and estimate error
22  $e_{n+1} = h \sum_{j=1}^3 d_j f_j$ 
23  $t_{n+1} := T_3$ 
24  $x_{n+1} := X_3$ 

```

7.1.3 The Van der Pol Problem

The ESDIRK23 method derived above was tested on the Van der Pol problem in (2.6). The same setup was used as in Section 6.1.3 with an initial step size $h_0 = 0.001$. The results can be seen in Figure 7.2 and Figure 7.3. Notice the higher values for this error estimator compared to the error estimators in previous chapters. For the stiff system, the steps can get very large. The method seems to be performing efficiently for both stiff and non-stiff systems. This is investigated further in the next section.

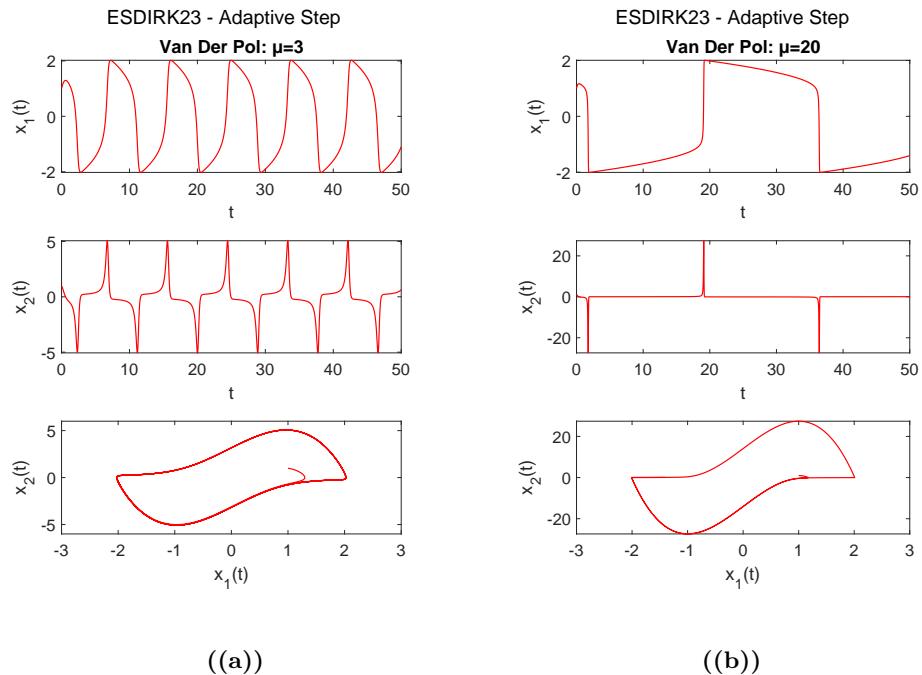


Figure 7.2: ESDIRK23 with adaptive step size on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.

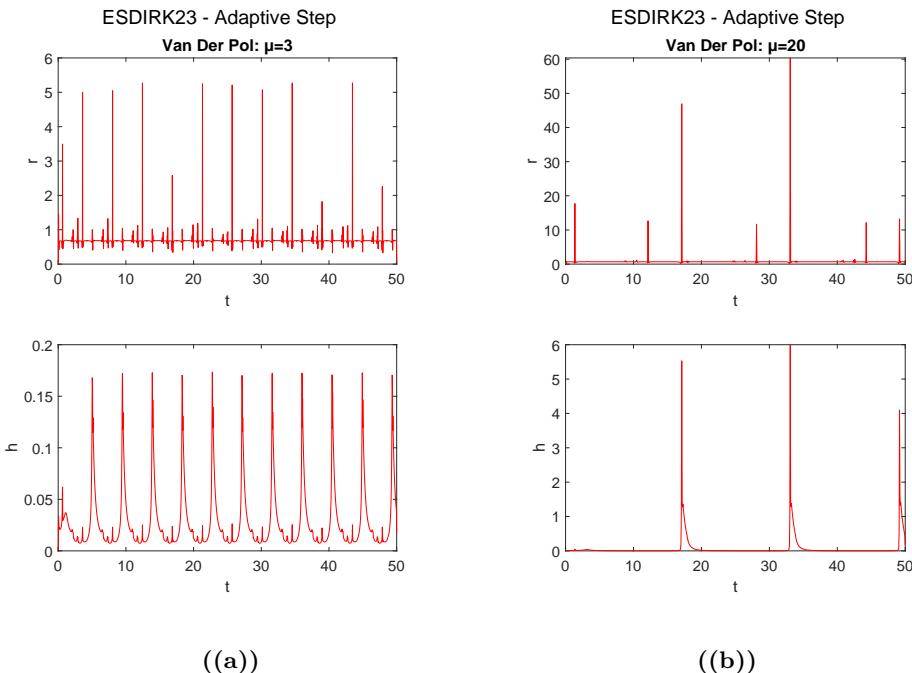


Figure 7.3: Error estimator, r , and step sizes, h , using ESDIRK23 with adaptive step size on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.

7.1.4 Comparing with Matlab's ODE Solvers and Implemented Solvers

To further test the performance of the ESDIRK23 method, it was compared with Matlab's ODE solvers, *ode45* and *ode15s*, and the other solvers developed in this exam problem. This was done for high and low tolerances, 10^{-3} and 10^{-7} on the Van der Pol problem as in the previous section. Table 7.3 shows that this method is performing slightly better computationally wise than *ode45* for the stiff problem with low tolerance. However, Table 7.4 shows that as the tolerance is decreased the function evaluations the method requires increases quite a bit. Looking at the approximations in Figure 7.4, the behaviour is the same for all three solvers. The method manages to maintain good accuracy even for the stiff system.

Table 7.3: Comparison of the ESDIRK23 method with Matlab's ODE solvers, *ode45* and *ode15s* for high tolerance of 10^{-3} .

	ESDIRK23		ode45		ode15s	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
N. Function	2651	1409	1489	3751	1393	750
N. Steps	487	232	248	625	675	363
N. Accepted steps	410	197	194	583	559	287
N. Rejected steps	77	35	54	42	116	76

Table 7.4: Comparison of the ESDIRK23 method with Matlab's ODE solvers, *ode45* and *ode15s* for low tolerance of 10^{-7} .

	ESDIRK23		ode45		ode15s	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
N. Function	34545	16001	6307	6769	4420	2325
N. Steps	8399	3800	1051	1128	2659	1274
N. Accepted steps	8397	3790	1024	1115	2505	1171
N. Rejected steps	2	10	27	13	154	103

Table 7.5 summarizes the performances of the methods implemented in this exam problem for a low tolerance of 10^{-7} . The DOPRI54 seems to be fastest for both stiff and non-stiff systems, with the classical Runge-Kutta and ESDIRK23 right behind. The Euler methods on the other hand are extremely slow. To visualize the performance difference, Figure 7.5 shows the solutions to the Van der Pol problem with high tolerance. The explicit Euler method seems to have the worst performance although it manages to maintain smoothness of the phase plot for both systems. The ESDIRK23 is slightly faster than the other methods in both cases. To conclude, the ESDIRK23 method manages to perform with good accuracy on the stiff Van der Pol system and performs quite well with lower tolerances.

Table 7.5: Comparing the performance of the methods implemented in the exam problem with adaptive step sizes, with tolerance 10^{-7} .

Method	N. Function		N. Accepted		N. Rejected	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
RK4	18132	15781	1412	1331	260	114
Explicit Euler	300800	134550	150400	67273	1	2
Implicit Euler	850730	380480	212680	95119	1	2
DOPRI54	7625	7636	887	994	236	113
ESDIRK23	34545	16001	8397	3790	2	10

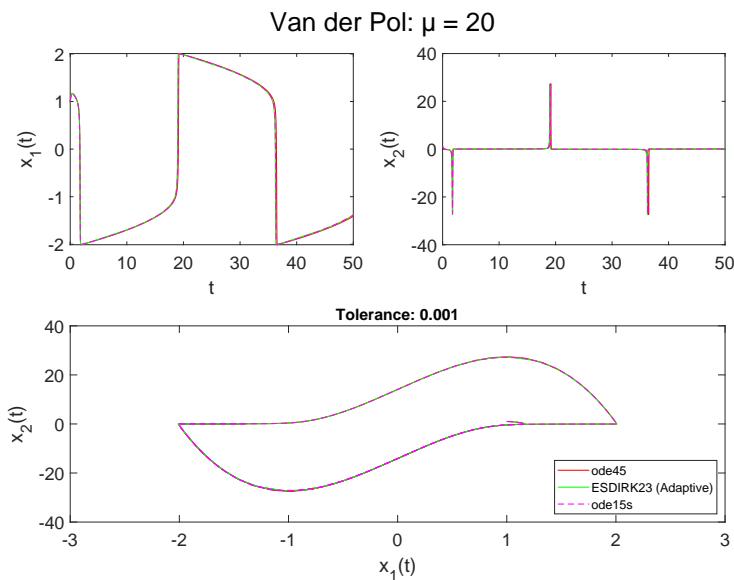
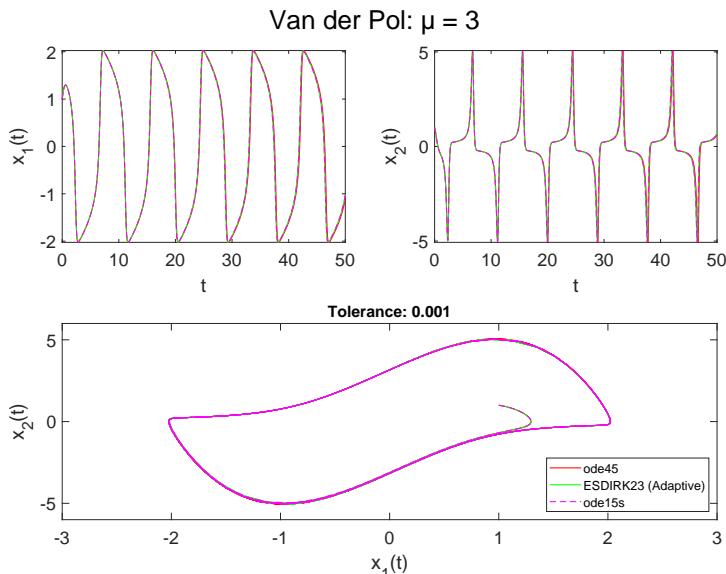


Figure 7.4: Comparison of ESDIRK23 with adaptive step size, *ode45* and *ode15s* on the Van der Pol problem with high tolerance.

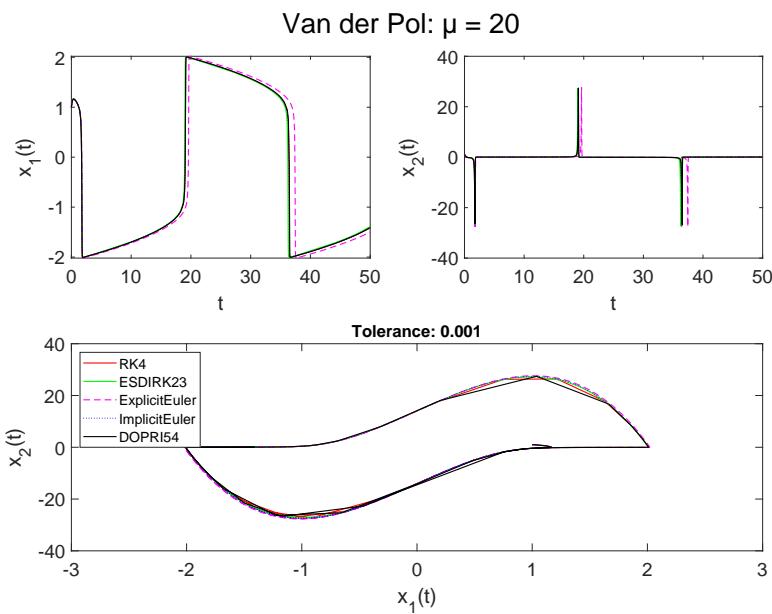
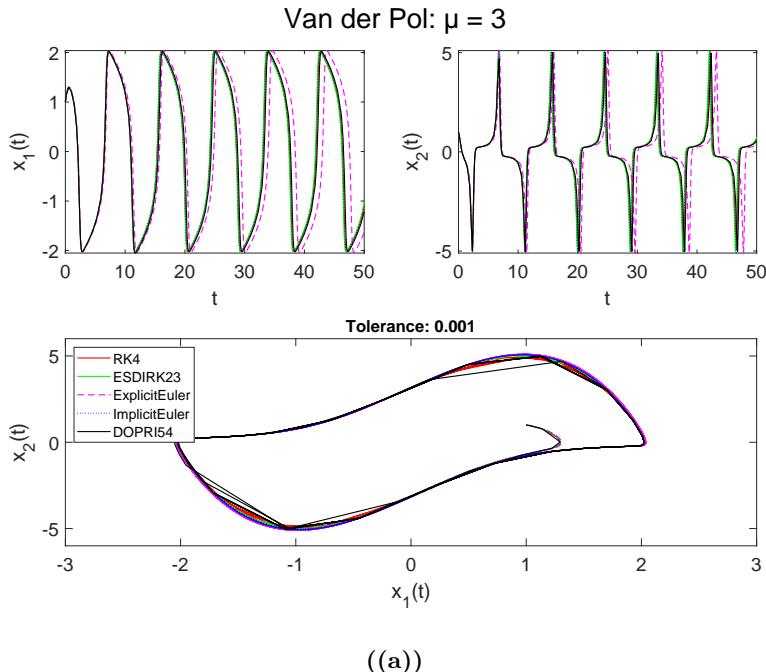


Figure 7.5: Comparison of ESDIRK23 with the other implemented solvers in this exam problem on the Van der Pol problem with high tolerance.

CHAPTER 8

Discussion and Conclusion

From the evaluations on the Van der Pol problem it can be concluded that the Euler methods perform best for higher tolerances and the Runge-Kutta methods for lower tolerances. This is due to the high amount of function evaluations required in the Euler methods for low tolerances. Modifying the methods to include adaptive steps enhanced their overall performance, with the implicit methods working well for the stiff problem and the explicit methods obtaining the best accuracy for the non-stiff problem. With regards to stability, the results revealed that the explicit Runge-Kutta and Euler methods do not satisfy the requirements for both A- and L-stability. However, the implicit methods, ESDIRK23 and implicit Euler, satisfy both.

APPENDIX A

An Appendix

A.1 Algorithms

A.1.1 Problem 2 Algorithms

```
1 function [T,X] = ExplicitEulerFixedStepSize(fun ,t0 ,tN,N,x0 ,varargin)
2 % EXPLICITEULERFIXEDSTEP SIZE Fixed step size Explicit Euler solver for
3 % systems of ODEs
4 %
5 % Syntax:
6 % [Tout ,Xout]=ExplicitEulerFixedStepSize(fun ,t0 ,tN,N,x0 ,varargin)
7 %
8 % Compute step size and allocate memory
9 dt = (tN-t0)/N;
10 nx = size(x0,1);
11 X = zeros(nx,N+1);
12 T = zeros(1,N+1);
13 %
14 % Eulers explicit method
15 T(:,1) = t0;
16 X(:,1) = x0;
17 for k=1:N
18     f = feval(fun , T(k) , X(:,k) , varargin{:});
19     T(:,k+1) = T(:,k) + dt;
20     X(:,k+1) = X(:,k) + f*dt;
21 end
22 %
23 T = T';
24 X = X';
```

Listing A.1: Explicit Euler Fixed Step Size

```
1 function [T,X,stat] = ExplicitEulerAdaptiveTimeStep(fun ,tspan ,x0 ,h0 ,abstol ,
2           reltol ,varargin)
3 % ExplicitEulerAdaptiveTimeStep Explicit Euler solver with adaptive step
4 % size and error estimation using step doubling for systems of ODEs
5 %
6 % Syntax:
```

```

6 % [T,X,stat] = ExplicitEulerAdaptiveTimeStep(fun ,tspan ,x0 ,h0 ,abstol ,reldtol ,
7 % varargin)
8 % Error controller parameters
9 epstol = 0.8; % target
10 facmin = 0.1; % maximum decrease factor
11 facmax = 5.0; % maximum increase factor
12
13 % Integration interval
14 t0 = tspan(1);
15 tf = tspan(2);
16 % Initial conditions
17 t = t0;
18 h = h0;
19 x = x0;
20
21 % Output;
22 T = t;
23 X = x';
24
25 % Store variables
26 stat.nfun = 0;
27 stat.nAccept = 0;
28 stat.nReject = 0;
29 stat.h = h;
30 stat.r = [];
31
32 % Main algorithm
33 while t < tf
34     if (t+h > tf)
35         h = tf - t;
36     end
37     f = feval(fun ,t ,x ,varargin{:});
38     stat.nfun = stat.nfun+1;
39
40     stat.h = [stat.h h];
41
42     AcceptStep = false;
43     while AcceptStep
44         % Take step of size h
45         x1 = x + h*f;
46
47         % Step Doubling for error estimation
48         % Take step of size h/2
49         hm = 0.5*h;
50         tm = t + hm;
51         xm = x + hm*f;
52         fm = feval(fun , tm,xm ,varargin{:});
53         stat.nfun = stat.nfun+1;
54         x1hat = xm + hm*fm;
55
56         % Error estimation
57         e = x1hat - x1;
58         r = max(abs(e)./max(abstol ,abs(x1hat).*reldtol));
59

```

```

60      AcceptStep = (r <= 1.0);
61      if AcceptStep
62
63          t = t+h;
64          x = x1hat; % advance most precise value of x
65
66          T = [T;t];
67          X = [X;x'];
68
69          stat.nAccept = stat.nAccept+1;
70          stat.r = [stat.r r];
71      else
72          stat.nReject = stat.nReject+1;
73
74      end
75      % Asymptotic step size controller
76      h = max(facmin, min(sqrt(epstol/r), facmax))*h;
77
78      stat.r = [stat.r r];
79  end
80 end
81 end

```

Listing A.2: Explicit Euler with adaptive step size and error estimation using step doubling.

A.1.2 Problem 3 Algorithms

```

1 function [T,X] = ImplicitEulerFixedStepSize(funJac,ta,tb,N,xa,varargin)
2 % ImplicitEulerFixedStepSize Implicit Euler solver for a fized step size
3 %
4 % funJac: Function to compute the objective function and it's Jacobian
5 % varargin: Parameters for the objective function
6 %
7 % Syntax: [T,X] = ImplicitEulerFixedStepSize(funJac,ta,tb,N,xa,varargin)
8
9 % Compute step size and allocate memory
10 dt = (tb-ta)/N;
11 nx = size(xa,1);
12 X = zeros(nx,N+1);
13 T = zeros(1,N+1);
14
15 tol = 1.0e-8;
16 maxit = 100;
17
18 % Eulers Implicit Method
19 T(:,1) = ta;
20 X(:,1) = xa;
21 for k=1:N
22     f = feval(funJac,T(k),X(:,k),varargin{:});
23     T(:,k+1) = T(:,k) + dt;

```

```

24 xinit = X(:,k) + f*dt;
25 X(:,k+1) = NewtonsMethodODE(funJac ,...
26 T(:,k), X(:,k), dt, xinit, tol, maxit, varargin{:});
27 end
28
29 % Form a nice table for the result
30 T = T';
31 X = X';

```

Listing A.3: Implicit Euler with fixed step size.

```

1 function [T,X,stat] = ImplicitEulerAdaptiveStep(funJac,tspan,x0,h0,abstol,
2      reltol,varargin)
% ImplicitEulerAdaptiveTimeStep Implicit Euler solver with adaptive step
3 % size and error estimation using step doubling for systems of ODEs
4 %
5 % funJac: Function to compute the objective function and it's Jacobian
6 % varargin: Parameters for the objective function
7 %
8 % Syntax:
9 % [T,X,stat] = ImplicitEulerAdaptiveStep(funJac,tspan,x0,h0,abstol,%
10 %     varargin)
11
11 epstol=0.8; %Target
12 facmin=0.1; %Maximum decrease factor
13 facmax=5.0; %Maximum increase factor
14
15 %Integration interval
16 t0=tspan(1);
17 tf=tspan(2);
18
19 %Initial Conditions
20 t=t0;
21 h=h0;
22 x=x0;
23
24 tol = 1.0e-8;
25 maxit = 100;
26
27 %Output
28 T=t;
29 X=x';
30
31 % Store variables
32 stat.nfun = 0;
33 stat.nAccept = 0;
34 stat.nReject = 0;
35 stat.h = h;
36 stat.r = [];
37
38 %Main algorithm
39 i=0;
40 while t<tf

```

```

41      i=i+1;
42      if (t+h>tf)
43          h= tf - t ;
44      end
45
46      stat.h = [ stat.h h];
47
48      [f J]=feval(funJac ,t ,x ,varargin{:});
49      stat.nfun = stat.nfun+1;
50
51      AcceptStep = false;
52
53      while AcceptStep
54          %Take step of size h
55          xinit=x+h*f;
56          x1=NewtonsMethodODE(funJac ,t , x , h, xinit , tol , maxit , varargin{:});
57
58          % Step Doubling for error estimation
59          %Take step of size h/2
60          hm=0.5*h;
61          tm=t+hm;
62          xminit=x+hm*f;
63          xm=NewtonsMethodODE(funJac ,t , x , hm, xminit , tol , maxit , varargin{:});
64          fm=feval(funJac ,tm,xm, varargin{:});
65          x1hat=xm+hm*fm;
66
67          %Error estimation
68          e=x1hat-x1;
69          r=max(abs(e)./max(abstol ,abs(x1hat).*reltol));
70
71          AcceptStep = (r<=1.0);
72          if AcceptStep
73              t=t+h;
74              x=x1hat;
75
76              T=[T; t ];
77              X=[X;x '];
78
79              stat.nAccept = stat.nAccept+1;
80              stat.r = [ stat.r r ];
81          end
82
83          %Asymptotic step size controller
84          h=max(facmin,min(sqrt(epstol/r),facmax))*h;
85
86          stat.r = [ stat.r r ];
87      end
88
89
90 end

```

Listing A.4: Implicit Euler with adaptive step size and error estimation using step doubling.

```

1 function [x,nfun] = NewtonsMethodODE(FunJac, tk, xk, dt, xinit, tol, maxit,
2 varargin)
% NewtonsMethodODE Performs Newton's method for approximating next step in
3 % the implicit Euler method
4 %
5 % Syntax: [x,nfun] = NewtonsMethodODE(FunJac,tk,xk,dt,xinit,tol,maxit,
6 % varargin)
7 k = 0;
8 t = tk + dt;
9 x = xinit;
10 [f,J] = feval(FunJac,t,x,varargin{:});
11 nfun = 1; % keep track of function evaluations
12
13 R = x - f*dt - xk;
14 I = eye(length(xk));
15 while( (k < maxit) && (norm(R, 'inf') > tol) )
16     k = k+1;
17     dRdx = I - J*dt;
18     dx = dRdx\R;
19     x = x - dx;
20     [f,J] = feval(FunJac,t,x,varargin{:});
21     nfun = nfun +1;
22     R = x - dt*f - xk;
23 end

```

Listing A.5: Newton's method for the next step in implicit Euler with adaptive step size.

A.1.3 Problem 4 Algorithms

```

1 function [W,Tw,dW] = StdWienerProcess(T,N,nW,Ns,seed)
2 % STDWIENERPROCESS Ns realizations of a standard Wiener process
3 %
4 % Syntax: [W,Tw,dW] = StdWienerProcess(T,N,Ns,seed)
5 %
6 % W : Standard Wiener process in [0,T]
7 % Tw : Time points
8 % dW : White noise used to generate the Wiener process
9 % T : Final time
10 % N : Number of intervals
11 % nW : Dimension of W(k)
12 % Ns : Number of realizations
13 % seed : To set the random number generator (optional)
14
15 if nargin == 4
16     rng(seed);
17 end
18 dt = T/N;
19 dW = sqrt(dt)*randn(nW,N,Ns);

```

```

20 W = [ zeros(nW,1 ,Ns) cumsum(dW,2) ];
21 Tw = 0:dt:T;
22 end

```

Listing A.6: A function to realize the standard Wiener process.

```

1 function X=SDEsolverExplicitExplicit(ffun ,gfun ,T,x0,W,varargin)
2 % SDEsolverExplicitExplicit Euler -Maruyama solver for systems of SDEs
3 %
4 % Syntax: X=SDEsolverExplicitExplicit (ffun ,gfun ,T,x0,W,varargin)
5
6 N = length(T);
7 nx = length(x0);
8 X = zeros(nx,N);
9
10 X(:,1) = x0;
11 for k=1:N-1
12     f = feval(ffun,T(k),X(:,k),varargin{:});
13     g = feval(gfun,T(k),X(:,k),varargin{:});
14     dt = T(k+1)-T(k);
15     dW = W(:,k+1)-W(:,k);
16     psi = X(:,k) + g*dW;
17     X(:,k+1) = psi + f*dt;
18 end
19 end

```

Listing A.7: The explicit-explicit method for solving systems of SDEs.

```

1 function X=SDEsolverImplicitExplicit(ffun ,gfun ,T,x0,W,varargin)
2 % SDEsolverImplicitExplicit Implicit - Explicit solver for
3 % systems of SDEs
4 %
5 % Syntax: X=SDEsolverImplicitExplicit (ffun ,gfun ,T,x0,W,varargin)
6
7 tol = 1.0e-8;
8 maxit = 100;
9
10 N = length(T);
11 nx = length(x0);
12 X = zeros(nx,N);
13
14 X(:,1) = x0;
15 k=1;
16 [f, ] = feval(ffun,T(k),X(:,k),varargin{:});
17
18 for k=1:N-1
19     g = feval(gfun,T(k),X(:,k),varargin{:});
20     dt = T(k+1)-T(k);
21     dW = W(:,k+1)-W(:,k);
22     psi = X(:,k) + g*dW;
23     xinit = psi + f*dt;

```

```

24 [X(:,k+1),f, ] = SDENewtonSolver ( ...
25 ffun ,...
26 T(:,k+1),dt,psi,xinit ,...
27 tol,maxit,varargin{:}); ...
28 end

```

Listing A.8: The implicit-explicit method for solving systems of SDEs.

A.1.4 Problem 5 Algorithms

```

1 function [Tout,Xout]=ClassicalRungeKuttaSolver(fun,tspan,x0,h,varargin)
2 % =ClassicalRungeKuttaSolver Fixed step size classical Runge-Kutta
3 % solver for systems of ODEs
4 %
5 % Syntax:
6 % [Tout,Xout]=ClassicalRungeKuttaSolver(fun,tspan,x0,h,varargin)
7 %
8 % Solver Parameters
9 s = 4; % Number of stages in ERK method
10 AT = [0,0,0,0;0.5,0,0,0;0,0.5,0,0;0,0,1,0]'; % Transpose of A-matrix in
   % Butcher tableau
11 b = [1/6,1/3,1/3,1/6]'; % b-vector in Butcher tableau
12 c = [0;1/2;1/2;1]; % c-vector in Butcher tableau
13 %
14 % Parameters related to constant step size
15 hAT = h*AT;
16 hb = h*b;
17 hc = h*c;
18 % Size parameters
19 x = x0;
20 t = tspan(1); % Initial time
21 tf = tspan(end); % Final time
22 N = (tf-t)/h; % Number of steps
23 nx = length(x0); % System size (dim(x))
24 % Allocate memory
25 T = zeros(1,s); % Stage T
26 X = zeros(nx,s); % Stage X
27 F = zeros(nx,s); % Stage F
28 Tout = zeros(N+1,1); % Time for output
29 Xout = zeros(N+1,nx); % States for output
30 %
31 Tout(1) = t;
32 Xout(1,:) = x';
33 for n=1:N
34   % Stage 1
35   T(1) = t;
36   X(:,1) = x;
37   F(:,1) = feval(fun,T(1),X(:,1),varargin{:});
38   % Stage 2,3,...,s
39   T(2:s) = t + hc(2:s);
40   for i=2:s

```

```

41      X(:, i) = x + F(:, 1:i-1)*hAT(1:i-1, i);
42      F(:, i) = feval(fun, T(i), X(:, i), varargin{:});
43  end
44 % Next step
45 t = t + h;
46 x = x + F*hb;
47 % Save output
48 Tout(n+1) = t;
49 Xout(n+1,:) = x';
50 end
51
52 end

```

Listing A.9: The classical Runge-Kutta method with fixed step size.

```

1 function [t1, x1] = ClassicalRungeKuttaStep(fun, t, x, f, h, varargin)
2
3 h2 = 0.5 * h;
4 alpha = h / 6;
5 beta = h / 3;
6
7 T1 = t;
8 X1 = x;
9 F1 = f;
10
11 T2 = x + h2;
12 X2 = x + h2 * F1;
13 F2 = feval(fun, T2, X2, varargin{:});
14
15 T3 = T2;
16 X3 = x + h2 * F2;
17 F3 = feval(fun, T3, X3, varargin{:});
18
19 T4 = t + h;
20 X4 = x + h * F3;
21 F4 = feval(fun, T4, X4, varargin{:});
22
23 t1 = T4;
24
25 x1 = x + alpha * (F1 + F4) + beta * (F2 + F3);
26
27 end

```

Listing A.10: One step of the classical Runge-Kutta method .

```

1 function [T, X, stat] = ClassicalRungeKuttaAdaptiveStep(fun, tspan, x0, h0, abstol,
2             reltol, varargin)
% ClassicalRungeKuttaAdaptiveStep Adaptive step size classical Runge-Kutta
3 % solver for systems of ODES with error estimation by step-doubling
4 %
5 % Syntax:

```

```

6 % [T,X,stat]=ClassicalRungeKuttaAdaptiveStep(fun ,tspan ,x0 ,h0 ,abstol ,reltol ,
7 %varargin )
8 %Error controller
9 epstol=0.8;
10 kpow=0.2;
11 facmin=0.1;
12 facmax=5;
13
14 %Integration interval
15 t0=tspan(1);
16 tf=tspan(2);
17
18 %Initial conditions
19 t=t0;
20 h=h0;
21 x=x0;
22
23 %Output
24 T=t ;
25 X=x' ;
26
27 % Storage variables
28 stat.nfun = 0;
29 stat.nAccept = 0;
30 stat.nReject = 0;
31 stat.h = [];
32 stat.r = [];
33
34 %Algorithm :
35
36
37 i=0;
38
39 while t<tf
40 i=i+1;
41
42 if (t+h>tf)
43 h=tf - t ;
44 end
45
46 stat.h = [stat.h h];
47
48 f=feval(fun ,t ,x ,varargin{:});
49 stat.nfun = stat.nfun +1;
50
51 AcceptStep=false ;
52
53 while AcceptStep
54 %take step of size h
55 [t1 ,x1]=ClassicalRungeKuttaStep(fun ,t ,x ,f ,h ,varargin{:});
56 stat.nfun = stat.nfun +3;
57
58 %take step of size h/2
59 hm=0.5*h;

```

```

60      [tm,xm]=ClassicalRungeKuttaStep(fun ,t ,x ,f ,hm, varargin {:}) ;
61      fm=feval(fun ,tm,xm, varargin {:}) ;
62      [t1hat ,x1hat]=ClassicalRungeKuttaStep(fun ,tm,xm,fm,hm, varargin {:}) ;
63      stat .nfun = stat .nfun +7;
64      %Error estimation
65      e=x1hat-xl ;
66      r=max(abs(e)./max(abstol ,abs(x1hat).*reltol));
67
68      AcceptStep = ( r <= 1.0) ;
69
70      if AcceptStep
71          t=t+h;
72          x=x1hat;
73
74          T=[T; t ];
75          X=[X;x '];
76          stat .nAccept = stat .nAccept+1;
77      else
78          stat .nReject = stat .nReject +1;
79      end
80      %Asymptotic step size controller
81      h=max(facmin ,min((epstol/r)^kpow ,facmax))*h;
82
83  end
84
85  %r_lis(i)=r;
86  stat .r = [stat .r r];
87
88 end

```

Listing A.11: The classical Runge-Kutta method with adaptive step size.

A.1.5 Problem 6 Algorithms

```

1 function [T,X,E,stat ] = DOPRI54AdaptiveStep(fun ,tspan ,x0 ,h0 ,abstol ,reltol ,
2 varargin)
% DOPRI54AdaptiveStep Implements the Dormand- Prince 54 method with
3 % adaptive step size and error estimation
4 %
5 % Syntax:
6 % [T,X,E,stat ] = DOPRI54AdaptiveStep(fun ,tspan ,x0 ,h0 ,abstol ,reltol ,varargin)
7
8 %
9 % Error controller
10 epstol = 0.8;
11 kpow = 0.2;
12 facmin = 0.1;
13 facmax = 5;
14
15 %Integration interval
16 t0 = tspan(1);

```

```

17 tf = tspan(2);
18
19 %Initial conditions
20 t=t0;
21 h=h0;
22 x=x0;
23
24 %Output
25 T=t;
26 X=x';
27 E=zeros(max(size(x)),1)';
28
29 % Storage variables
30 stat.nfun = 0;
31 stat.r = [];
32 stat.nAccept = 0;
33 stat.nReject = 0;
34 stat.h = [];
35
36 while t < tf
37   if(t+h>tf)
38     h = tf-t;
39   end
40   f = feval(fun,t,x,varargin{:});
41   stat.nfun = stat.nfun + 1;
42
43   stat.h = [stat.h h];
44
45 AcceptStep=false;
46 while AcceptStep
47   %Take step of size h with built in error estimation
48   [t1,x1,e] = DOPRI54Step(fun,t,x,f,h,varargin{:});
49   stat.nfun = stat.nfun+6;
50
51   %Error estimation
52   r = max(abs(e)./max(abstol,abs(x1).*reltol));
53
54   AcceptStep = (r <= 1.0);
55   if AcceptStep
56     t = t1;
57     x = x1;
58
59     T = [T;t];
60     X = [X;x'];
61     E = [E;e'];
62
63     stat.nAccept = stat.nAccept+1;
64   else
65     stat.nReject = stat.nReject+1;
66   end
67   %Asymptotic step size controller
68   h = max(facmin,min((epstol/r)^kpow,facmax))*h;
69 end
70 stat.r = [stat.r r];
71 end

```

Listing A.12: The DOPRI54 method with adaptive step size.

```

1 function [t1,x1,e] = DOPRI54Step(fun,t,x,f,h,varargin)
2
3 s = 7;
4 A = zeros(s,s);
5 A(2,1) = 1/5;
6 A(3,1) = 3/40;
7 A(4,1) = 44/45;
8 A(5,1) = 19372/6561;
9 A(6,1) = 9017/3168;
10 A(7,1) = 35/384;
11 A(3,2) = 9/40;
12 A(4,2) = -56/15;
13 A(5,2) = -25360/2187;
14 A(6,2) = -355/33;
15 A(4,3) = 32/9;
16 A(5,3) = 64448/6561;
17 A(6,3) = 46732/5247;
18 A(7,3) = 500/1113;
19 A(5,4) = -212/729;
20 A(6,4) = 49/176;
21 A(7,4) = 125/192;
22 A(6,5) = -5103/18656;
23 A(7,5) = -2187/6784;
24 A(7,6) = 11/84;
25
26 b = [35/384; 0; 500/1113; 125/192; -2187/6784; 11/84; 0];
27 c = [0; 1/5; 3/10; 4/5; 8/9; 1; 1];
28 d = [71/57600; 0; -71/16695; 71/1920; -17253/339200; 22/525; -1/40];
29
30 T1 = t;
31 X1 = x;
32 F1 = f;
33
34 X2 = x+A(2,1)*h*f;
35 F2 = feval(fun,t+c(2)*h,X2,varargin{:});
36
37 X3 = x+h*(A(3,1)*f+A(3,2)*F2);
38 F3 = feval(fun,t+c(3)*h,X3,varargin{:});
39
40 X4 = x+h*(A(4,1)*f + A(4,2)*F2+A(4,3)*F3);
41 F4 = feval(fun,t+c(4)*h,X4,varargin{:});
42
43 X5 = x+h*(A(5,1)*f+A(5,2)*F2+A(5,3)*F3+A(5,4)*F4);
44 F5 = feval(fun,t+c(5)*h,X5,varargin{:});
45
46 X6 = x+h*(A(6,1)*f+A(6,2)*F2+A(6,3)*F3+A(6,4)*F4+A(6,5)*F5);
47 F6 = feval(fun,t+c(6),X6,varargin{:});
48
49 X7 = x+h*(A(7,1)*f+A(7,3)*F3+A(7,4)*F4+A(7,5)*F5+A(7,6)*F6);
50 F7 = feval(fun,t+c(7),X7,varargin{:});

```

```

51 %Output
52 e = h*(d(1)*f+d(3)*F3+d(4)*F4+d(5)*F5+d(6)*F6+d(7)*F7);
53 t1 = t+c(7)*h;
54 x1 = X7;

```

Listing A.13: One step of the DOPRI54 method.

```

1 function [Tdot] = CSTR_1D(t,T,flowrate)
2 % CSTR_1D
3 %
4 % Syntax: [Tdot] = CSTR_1D(t,T,flowrate)
5 %
6 % Parameters given in CSTR presentation from lecture notes
7 rho = 1.0;
8 cP = 4.186;
9 k0 = exp(24.6)*60; % in minutes
10 EaR = 8500; % Ea/R
11 DeltaH = -560; % Reaction enthalpy
12 V = 0.105; %1.0; %0.105;
13 CAin = 1.6/2;
14 CBin = 2.4/2;
15 Tin = 273.65; % in Kelvin
16
17 beta = -DeltaH/(rho*cP);
18
19 % Get flow rate trajectory
20 F = feval(flowrate, t);
21 F = F/1000; % Convert mL to L
22
23 % Approximations
24 CA = CAin + 1/beta*(Tin-T);
25 CB = CBin + 2/beta*(Tin-T);
26
27 % Reaction rate
28 kT = k0*exp(-EaR*(1/T)); % Ea/(R*T)
29 r = kT*CA*CB; % (r(CA,CB,T))
30
31 RT = beta*r; % Production rate of temperature
32 % Mass and energy balance considerations of the constant volume reactor:
33 Tdot = F/V*(Tin-T) + RT;
34 end
35
36
37 function [xdot] = CSTR_3D(t,x,flowrate)
38 % CSTR_3D
39 %
40 % Syntax: [Tdot,CAdot,CBdot] = CSTR_1D(t,x,V,CAin,CBin,Tin,beta,EaR,k0)
41 %
42 % unpack T variable
43 T = x(3,1);
44
45 % Parameters given in CSTR presentation from lecture notes

```

```
46 rho = 1.0;
47 cP = 4.186;
48 k0 = exp(24.6)*60; % in minutes
49 EaR = 8500; % Ea/R
50 DeltaH = -560; % Reaction enthalpy
51 V = 0.105;
52 CAin = 1.6/2;
53 CBin = 2.4/2;
54 Tin = 273.65; % in Kelvin
55 %Tin = 0.5; % in Celsius
56
57 beta = -DeltaH/(rho*cP);
58
59 % Get flow rate trajectory
60 F = feval(flowrate, t);
61 F = F/1000; % Convert mL to L
62
63 % Approximations
64 CA = CAin + 1/beta*(Tin-T);
65 CB = CBin + 2/beta*(Tin-T);
66
67 % Reaction rate
68 kT = k0*exp(-EaR*(1/T)); % Ea/(R*T)
69 r = kT*CA*CB; % (r(CA,CB,T))
70
71 RT = beta*r; % Production rate of temperature
72 RA = -r; % Production rate of A
73 RB = -2*r; % Production rate of B
74
75 % Mass and energy balance considerations of the constant volume reactor:
76 CAdot = F/V*(CAin-x(1,1)) + RA;
77 CBdot = F/V*(CBin-x(2,1)) + RB;
78 Tdot = F/V*(Tin-T) + RT;
79
80 xdot = [CAdot, CBdot, Tdot]';
81
82 end
83
84 function [F] = FlowRate(t)
85 % FlowRate Computes the flow rate trajectory in mL/min given time t
86 %
87 % Syntax: [F] = FlowRate(t)
88
89 if t <= 3.5
90     F = 700;
91     return;
92 end
93 if t <= 5
94     F = 600;
95     return;
96 end
97 if t <= 9
98     F = 400;
99     return;
100 end
```

```

101 if t <= 12
102     F = 300;
103     return;
104 end
105 if t <= 16
106     F = 200;
107     return;
108 end
109 if t <= 18
110     F = 300;
111     return;
112 end
113 if t <= 20
114     F = 400;
115     return;
116 end
117 if t <= 22
118     F = 500;
119     return;
120 end
121 if t <= 24
122     F = 600;
123     return;
124 end
125 if t <= 28
126     F = 700;
127     return;
128 end
129 if t <= 32
130     F = 200;
131     return;
132 end
133 if t <= 35
134     F = 700;
135     return;
136 end
137
138 F = 0;
139
140 end

```

Listing A.14: The CSTR functions 1D and 3D along with the flow rate function

A.1.6 Problem 7 Algorithms

```

1 function [Tout,Xout,Gout,info ,stats] = ESDIRK23AdaptiveStep(fun ,jac ,tspan ,x0
2 ,h0,absTol,relTol ,varargin)
3 % ESDIRK23AdaptiveStep ESDIRK23 solver with adaptive step size
4 % for solution of ODEs
5 % Syntax:

```

```

6 % [Tout,Xout,Gout,info ,stats] = ESDIRK23AdaptiveStep(fun ,jac ,tspan ,x0,h0 ,
7   absTol ,relTol ,varargin )
8
9 %ESDIRK23 parameters
10 gamma = 1-1/sqrt(2);
11 a31 = (1-gamma)/2;
12 AT = [0 gamma a31;0 gamma a31;0 0 gamma];
13 c = [0; 2*gamma; 1];
14 b = AT(:,3);
15 bhat = [ (6*gamma-1)/(12*gamma); ...
16   1/(12*gamma*(1+2*gamma)); ...
17   (1-3*gamma)/(3*(1-2*gamma)) ];
18 d = b-bhat;
19 p = 2;
20 phat = 3;
21 s = 3;
22 t0 = tspan(1);
23 tf = tspan(2);
24
25 % Error and convergence controller
26 epsilon = 0.8;
27 tau = 0.1*epsilon;
28 itermax = 20;
29 ke0 = 1.0/phat;
30 ke1 = 1.0/phat;
31 ke2 = 1.0/phat;
32 alpharef = 0.3;
33 alphaJac = -0.2;
34 alphaLU = -0.2;
35 hrmin = 0.01;
36 hrmax = 10;
37 info = struct('nStage',s, ...
38   'absTol','dummy', ...
39   'relTol','dummy', ...
40   'iterMax',itermax, ...
41   'tspan',tspan, ...
42   'nFun',0, ...
43   'nJac',0, ...
44   'nLU',0, ...
45   'nBack',0, ...
46   'nStep',0, ...
47   'nAccept',0, ...
48   'nFail',0, ...
49   'nDiverge',0, ...
50   'nSlowConv',0);
51
52 %% Main integrator
53 nx = size(x0,1);
54 F = zeros(nx,s);
55 t = t0;
56 x = x0;
57 h = h0;
58
59 [F(:,1),g] = feval(fun,t,x,varargin{:});

```

```

50 info.nFun = info.nFun+1;
51 [dfdx,dgdx] = feval(jac,t,x,varargin{:});
52 info.nJac = info.nJac+1;
53 FreshJacobian = true;
54 if (t+h)>tf
55     h = tf-t;
56 end
57 hgammma = h*gamma;
58 dRdx = dgdx - hgammma*dfdx;
59 [L,U,pivot] = lu(dRdx, 'vector');
60 info.nLU = info.nLU+1;
61 hLU = h;
62
63 FirstStep = true;
64 ConvergenceRestriction = false;
65 PreviousReject = false;
66 iter = zeros(1,s);
67
68 % Initialize output
69 Nsize = 100;
70 Tout = zeros(Nsize,1);
71 Xout = zeros(Nsize,nx);
72 Gout = zeros(Nsize,nx);
73
74 Tout(1,1) = t;
75 Xout(1,:) = x.';
76 Gout(1,:) = g.';
77
78 while t<tf
79     info.nStep = info.nStep+1;
80
81     % ESDIRK23
82     i=1;
83     diverging = false;
84     SlowConvergence = false; % carsten
85     alpha = 0.0;
86     Converged = true;
87     while (i<s) && Converged
88         % Stage i=2,...,s of the ESDIRK Method
89         i=i+1;
90         phi = g + F(:,1:i-1)*(h*AT(1:i-1,i));
91
92         % Initial guess for the state
93         if i==2
94             dt = c(i)*h;
95             G = g + dt*F(:,1);
96             X = x + dgdx\G;
97         else
98             dt = c(i)*h;
99             G = g + dt*F(:,1);
100            X = x + dgdx\G;
101        end
102        T = t+dt;
103
104        [F(:,i),G] = feval(fun,T,X,varargin{:});
105
106        if abs(G-g) < 1e-10
107            Converged = true;
108        else
109            Converged = false;
110        end
111    end
112
113    if Converged
114        % Check convergence restriction
115        if abs(G-g) > 1e-10
116            Converged = false;
117        end
118    end
119
120    if Converged
121        % Update state
122        x = X;
123        g = G;
124        t = T;
125    else
126        % Revert to previous state
127        x = Xold;
128        g = Gold;
129        t = Told;
130        Converged = false;
131    end
132
133    if Converged
134        % Check slow convergence
135        if abs(G-g) > 1e-10
136            Converged = false;
137        end
138    end
139
140    if Converged
141        % Check divergence
142        if abs(G-g) > 1e-10
143            Converged = false;
144        end
145    end
146
147    if Converged
148        % Check previous reject
149        if PreviousReject
150            Converged = false;
151        end
152    end
153
154    if Converged
155        % Check first step
156        if FirstStep
157            Converged = false;
158        end
159    end
160
161    if Converged
162        % Check slow convergence
163        if SlowConvergence
164            Converged = false;
165        end
166    end
167
168    if Converged
169        % Check divergence
170        if diverging
171            Converged = false;
172        end
173    end
174
175    if Converged
176        % Check previous reject
177        if PreviousReject
178            Converged = false;
179        end
180    end
181
182    if Converged
183        % Check first step
184        if FirstStep
185            Converged = false;
186        end
187    end
188
189    if Converged
190        % Check slow convergence
191        if SlowConvergence
192            Converged = false;
193        end
194    end
195
196    if Converged
197        % Check divergence
198        if diverging
199            Converged = false;
200        end
201    end
202
203    if Converged
204        % Check previous reject
205        if PreviousReject
206            Converged = false;
207        end
208    end
209
210    if Converged
211        % Check first step
212        if FirstStep
213            Converged = false;
214        end
215    end
216
217    if Converged
218        % Check slow convergence
219        if SlowConvergence
220            Converged = false;
221        end
222    end
223
224    if Converged
225        % Check divergence
226        if diverging
227            Converged = false;
228        end
229    end
230
231    if Converged
232        % Check previous reject
233        if PreviousReject
234            Converged = false;
235        end
236    end
237
238    if Converged
239        % Check first step
240        if FirstStep
241            Converged = false;
242        end
243    end
244
245    if Converged
246        % Check slow convergence
247        if SlowConvergence
248            Converged = false;
249        end
250    end
251
252    if Converged
253        % Check divergence
254        if diverging
255            Converged = false;
256        end
257    end
258
259    if Converged
260        % Check previous reject
261        if PreviousReject
262            Converged = false;
263        end
264    end
265
266    if Converged
267        % Check first step
268        if FirstStep
269            Converged = false;
270        end
271    end
272
273    if Converged
274        % Check slow convergence
275        if SlowConvergence
276            Converged = false;
277        end
278    end
279
280    if Converged
281        % Check divergence
282        if diverging
283            Converged = false;
284        end
285    end
286
287    if Converged
288        % Check previous reject
289        if PreviousReject
290            Converged = false;
291        end
292    end
293
294    if Converged
295        % Check first step
296        if FirstStep
297            Converged = false;
298        end
299    end
300
301    if Converged
302        % Check slow convergence
303        if SlowConvergence
304            Converged = false;
305        end
306    end
307
308    if Converged
309        % Check divergence
310        if diverging
311            Converged = false;
312        end
313    end
314
315    if Converged
316        % Check previous reject
317        if PreviousReject
318            Converged = false;
319        end
320    end
321
322    if Converged
323        % Check first step
324        if FirstStep
325            Converged = false;
326        end
327    end
328
329    if Converged
330        % Check slow convergence
331        if SlowConvergence
332            Converged = false;
333        end
334    end
335
336    if Converged
337        % Check divergence
338        if diverging
339            Converged = false;
340        end
341    end
342
343    if Converged
344        % Check previous reject
345        if PreviousReject
346            Converged = false;
347        end
348    end
349
350    if Converged
351        % Check first step
352        if FirstStep
353            Converged = false;
354        end
355    end
356
357    if Converged
358        % Check slow convergence
359        if SlowConvergence
360            Converged = false;
361        end
362    end
363
364    if Converged
365        % Check divergence
366        if diverging
367            Converged = false;
368        end
369    end
370
371    if Converged
372        % Check previous reject
373        if PreviousReject
374            Converged = false;
375        end
376    end
377
378    if Converged
379        % Check first step
380        if FirstStep
381            Converged = false;
382        end
383    end
384
385    if Converged
386        % Check slow convergence
387        if SlowConvergence
388            Converged = false;
389        end
390    end
391
392    if Converged
393        % Check divergence
394        if diverging
395            Converged = false;
396        end
397    end
398
399    if Converged
400        % Check previous reject
401        if PreviousReject
402            Converged = false;
403        end
404    end
405
406    if Converged
407        % Check first step
408        if FirstStep
409            Converged = false;
410        end
411    end
412
413    if Converged
414        % Check slow convergence
415        if SlowConvergence
416            Converged = false;
417        end
418    end
419
420    if Converged
421        % Check divergence
422        if diverging
423            Converged = false;
424        end
425    end
426
427    if Converged
428        % Check previous reject
429        if PreviousReject
430            Converged = false;
431        end
432    end
433
434    if Converged
435        % Check first step
436        if FirstStep
437            Converged = false;
438        end
439    end
440
441    if Converged
442        % Check slow convergence
443        if SlowConvergence
444            Converged = false;
445        end
446    end
447
448    if Converged
449        % Check divergence
450        if diverging
451            Converged = false;
452        end
453    end
454
455    if Converged
456        % Check previous reject
457        if PreviousReject
458            Converged = false;
459        end
460    end
461
462    if Converged
463        % Check first step
464        if FirstStep
465            Converged = false;
466        end
467    end
468
469    if Converged
470        % Check slow convergence
471        if SlowConvergence
472            Converged = false;
473        end
474    end
475
476    if Converged
477        % Check divergence
478        if diverging
479            Converged = false;
480        end
481    end
482
483    if Converged
484        % Check previous reject
485        if PreviousReject
486            Converged = false;
487        end
488    end
489
490    if Converged
491        % Check first step
492        if FirstStep
493            Converged = false;
494        end
495    end
496
497    if Converged
498        % Check slow convergence
499        if SlowConvergence
500            Converged = false;
501        end
502    end
503
504    if Converged
505        % Check divergence
506        if diverging
507            Converged = false;
508        end
509    end
510
511    if Converged
512        % Check previous reject
513        if PreviousReject
514            Converged = false;
515        end
516    end
517
518    if Converged
519        % Check first step
520        if FirstStep
521            Converged = false;
522        end
523    end
524
525    if Converged
526        % Check slow convergence
527        if SlowConvergence
528            Converged = false;
529        end
530    end
531
532    if Converged
533        % Check divergence
534        if diverging
535            Converged = false;
536        end
537    end
538
539    if Converged
540        % Check previous reject
541        if PreviousReject
542            Converged = false;
543        end
544    end
545
546    if Converged
547        % Check first step
548        if FirstStep
549            Converged = false;
550        end
551    end
552
553    if Converged
554        % Check slow convergence
555        if SlowConvergence
556            Converged = false;
557        end
558    end
559
560    if Converged
561        % Check divergence
562        if diverging
563            Converged = false;
564        end
565    end
566
567    if Converged
568        % Check previous reject
569        if PreviousReject
570            Converged = false;
571        end
572    end
573
574    if Converged
575        % Check first step
576        if FirstStep
577            Converged = false;
578        end
579    end
580
581    if Converged
582        % Check slow convergence
583        if SlowConvergence
584            Converged = false;
585        end
586    end
587
588    if Converged
589        % Check divergence
590        if diverging
591            Converged = false;
592        end
593    end
594
595    if Converged
596        % Check previous reject
597        if PreviousReject
598            Converged = false;
599        end
600    end
601
602    if Converged
603        % Check first step
604        if FirstStep
605            Converged = false;
606        end
607    end
608
609    if Converged
610        % Check slow convergence
611        if SlowConvergence
612            Converged = false;
613        end
614    end
615
616    if Converged
617        % Check divergence
618        if diverging
619            Converged = false;
620        end
621    end
622
623    if Converged
624        % Check previous reject
625        if PreviousReject
626            Converged = false;
627        end
628    end
629
630    if Converged
631        % Check first step
632        if FirstStep
633            Converged = false;
634        end
635    end
636
637    if Converged
638        % Check slow convergence
639        if SlowConvergence
640            Converged = false;
641        end
642    end
643
644    if Converged
645        % Check divergence
646        if diverging
647            Converged = false;
648        end
649    end
650
651    if Converged
652        % Check previous reject
653        if PreviousReject
654            Converged = false;
655        end
656    end
657
658    if Converged
659        % Check first step
660        if FirstStep
661            Converged = false;
662        end
663    end
664
665    if Converged
666        % Check slow convergence
667        if SlowConvergence
668            Converged = false;
669        end
670    end
671
672    if Converged
673        % Check divergence
674        if diverging
675            Converged = false;
676        end
677    end
678
679    if Converged
680        % Check previous reject
681        if PreviousReject
682            Converged = false;
683        end
684    end
685
686    if Converged
687        % Check first step
688        if FirstStep
689            Converged = false;
690        end
691    end
692
693    if Converged
694        % Check slow convergence
695        if SlowConvergence
696            Converged = false;
697        end
698    end
699
700    if Converged
701        % Check divergence
702        if diverging
703            Converged = false;
704        end
705    end
706
707    if Converged
708        % Check previous reject
709        if PreviousReject
710            Converged = false;
711        end
712    end
713
714    if Converged
715        % Check first step
716        if FirstStep
717            Converged = false;
718        end
719    end
720
721    if Converged
722        % Check slow convergence
723        if SlowConvergence
724            Converged = false;
725        end
726    end
727
728    if Converged
729        % Check divergence
730        if diverging
731            Converged = false;
732        end
733    end
734
735    if Converged
736        % Check previous reject
737        if PreviousReject
738            Converged = false;
739        end
740    end
741
742    if Converged
743        % Check first step
744        if FirstStep
745            Converged = false;
746        end
747    end
748
749    if Converged
750        % Check slow convergence
751        if SlowConvergence
752            Converged = false;
753        end
754    end
755
756    if Converged
757        % Check divergence
758        if diverging
759            Converged = false;
760        end
761    end
762
763    if Converged
764        % Check previous reject
765        if PreviousReject
766            Converged = false;
767        end
768    end
769
770    if Converged
771        % Check first step
772        if FirstStep
773            Converged = false;
774        end
775    end
776
777    if Converged
778        % Check slow convergence
779        if SlowConvergence
780            Converged = false;
781        end
782    end
783
784    if Converged
785        % Check divergence
786        if diverging
787            Converged = false;
788        end
789    end
790
791    if Converged
792        % Check previous reject
793        if PreviousReject
794            Converged = false;
795        end
796    end
797
798    if Converged
799        % Check first step
800        if FirstStep
801            Converged = false;
802        end
803    end
804
805    if Converged
806        % Check slow convergence
807        if SlowConvergence
808            Converged = false;
809        end
810    end
811
812    if Converged
813        % Check divergence
814        if diverging
815            Converged = false;
816        end
817    end
818
819    if Converged
820        % Check previous reject
821        if PreviousReject
822            Converged = false;
823        end
824    end
825
826    if Converged
827        % Check first step
828        if FirstStep
829            Converged = false;
830        end
831    end
832
833    if Converged
834        % Check slow convergence
835        if SlowConvergence
836            Converged = false;
837        end
838    end
839
840    if Converged
841        % Check divergence
842        if diverging
843            Converged = false;
844        end
845    end
846
847    if Converged
848        % Check previous reject
849        if PreviousReject
850            Converged = false;
851        end
852    end
853
854    if Converged
855        % Check first step
856        if FirstStep
857            Converged = false;
858        end
859    end
860
861    if Converged
862        % Check slow convergence
863        if SlowConvergence
864            Converged = false;
865        end
866    end
867
868    if Converged
869        % Check divergence
870        if diverging
871            Converged = false;
872        end
873    end
874
875    if Converged
876        % Check previous reject
877        if PreviousReject
878            Converged = false;
879        end
880    end
881
882    if Converged
883        % Check first step
884        if FirstStep
885            Converged = false;
886        end
887    end
888
889    if Converged
890        % Check slow convergence
891        if SlowConvergence
892            Converged = false;
893        end
894    end
895
896    if Converged
897        % Check divergence
898        if diverging
899            Converged = false;
900        end
901    end
902
903    if Converged
904        % Check previous reject
905        if PreviousReject
906            Converged = false;
907        end
908    end
909
910    if Converged
911        % Check first step
912        if FirstStep
913            Converged = false;
914        end
915    end
916
917    if Converged
918        % Check slow convergence
919        if SlowConvergence
920            Converged = false;
921        end
922    end
923
924    if Converged
925        % Check divergence
926        if diverging
927            Converged = false;
928        end
929    end
930
931    if Converged
932        % Check previous reject
933        if PreviousReject
934            Converged = false;
935        end
936    end
937
938    if Converged
939        % Check first step
940        if FirstStep
941            Converged = false;
942        end
943    end
944
945    if Converged
946        % Check slow convergence
947        if SlowConvergence
948            Converged = false;
949        end
950    end
951
952    if Converged
953        % Check divergence
954        if diverging
955            Converged = false;
956        end
957    end
958
959    if Converged
960        % Check previous reject
961        if PreviousReject
962            Converged = false;
963        end
964    end
965
966    if Converged
967        % Check first step
968        if FirstStep
969            Converged = false;
970        end
971    end
972
973    if Converged
974        % Check slow convergence
975        if SlowConvergence
976            Converged = false;
977        end
978    end
979
980    if Converged
981        % Check divergence
982        if diverging
983            Converged = false;
984        end
985    end
986
987    if Converged
988        % Check previous reject
989        if PreviousReject
990            Converged = false;
991        end
992    end
993
994    if Converged
995        % Check first step
996        if FirstStep
997            Converged = false;
998        end
999    end
1000
1001    if Converged
1002        % Check slow convergence
1003        if SlowConvergence
1004            Converged = false;
1005        end
1006    end
1007
1008    if Converged
1009        % Check divergence
1010        if diverging
1011            Converged = false;
1012        end
1013    end
1014
1015    if Converged
1016        % Check previous reject
1017        if PreviousReject
1018            Converged = false;
1019        end
1020    end
1021
1022    if Converged
1023        % Check first step
1024        if FirstStep
1025            Converged = false;
1026        end
1027    end
1028
1029    if Converged
1030        % Check slow convergence
1031        if SlowConvergence
1032            Converged = false;
1033        end
1034    end
1035
1036    if Converged
1037        % Check divergence
1038        if diverging
1039            Converged = false;
1040        end
1041    end
1042
1043    if Converged
1044        % Check previous reject
1045        if PreviousReject
1046            Converged = false;
1047        end
1048    end
1049
1050    if Converged
1051        % Check first step
1052        if FirstStep
1053            Converged = false;
1054        end
1055    end
1056
1057    if Converged
1058        % Check slow convergence
1059        if SlowConvergence
1060            Converged = false;
1061        end
1062    end
1063
1064    if Converged
1065        % Check divergence
1066        if diverging
1067            Converged = false;
1068        end
1069    end
1070
1071    if Converged
1072        % Check previous reject
1073        if PreviousReject
1074            Converged = false;
1075        end
1076    end
1077
1078    if Converged
1079        % Check first step
1080        if FirstStep
1081            Converged = false;
1082        end
1083    end
1084
1085    if Converged
1086        % Check slow convergence
1087        if SlowConvergence
1088            Converged = false;
1089        end
1090    end
1091
1092    if Converged
1093        % Check divergence
1094        if diverging
1095            Converged = false;
1096        end
1097    end
1098
1099    if Converged
1100        % Check previous reject
1101        if PreviousReject
1102            Converged = false;
1103        end
1104    end
1105
1106    if Converged
1107        % Check first step
1108        if FirstStep
1109            Converged = false;
1110        end
1111    end
1112
1113    if Converged
1114        % Check slow convergence
1115        if SlowConvergence
1116            Converged = false;
1117        end
1118    end
1119
1120    if Converged
1121        % Check divergence
1122        if diverging
1123            Converged = false;
1124        end
1125    end
1126
1127    if Converged
1128        % Check previous reject
1129        if PreviousReject
1130            Converged = false;
1131        end
1132    end
1133
1134    if Converged
1135        % Check first step
1136        if FirstStep
1137            Converged = false;
1138        end
1139    end
1140
1141    if Converged
1142        % Check slow convergence
1143        if SlowConvergence
1144            Converged = false;
1145        end
1146    end
1147
1148    if Converged
1149        % Check divergence
1150        if diverging
1151            Converged = false;
1152        end
1153    end
1154
1155    if Converged
1156        % Check previous reject
1157        if PreviousReject
1158            Converged = false;
1159        end
1160    end
1161
1162    if Converged
1163        % Check first step
1164        if FirstStep
1165            Converged = false;
1166        end
1167    end
1168
1169    if Converged
1170        % Check slow convergence
1171        if SlowConvergence
1172            Converged = false;
1173        end
1174    end
1175
1176    if Converged
1177        % Check divergence
1178        if diverging
1179            Converged = false;
1180        end
1181    end
1182
1183    if Converged
1184        % Check previous reject
1185        if PreviousReject
1186            Converged = false;
1187        end
1188    end
1189
1190    if Converged
1191        % Check first step
1192        if FirstStep
1193            Converged = false;
1194        end
1195    end
1196
1197    if Converged
1198        % Check slow convergence
1199        if SlowConvergence
1200            Converged = false;
1201        end
1202    end
1203
1204    if Converged
1205        % Check divergence
1206        if diverging
1207            Converged = false;
1208        end
1209    end
1210
1211    if Converged
1212        % Check previous reject
1213        if PreviousReject
1214            Converged = false;
1215        end
1216    end
1217
1218    if Converged
1219        % Check first step
1220        if FirstStep
1221            Converged = false;
1222        end
1223    end
1224
1225    if Converged
1226        % Check slow convergence
1227        if SlowConvergence
1228            Converged = false;
1229        end
1230    end
1231
1232    if Converged
1233        % Check divergence
1234        if diverging
1235            Converged = false;
1236        end
1237    end
1238
1239    if Converged
1240        % Check previous reject
1241        if PreviousReject
1242            Converged = false;
1243        end
1244    end
1245
1246    if Converged
1247        % Check first step
1248        if FirstStep
1249            Converged = false;
1250        end
1251    end
1252
1253    if Converged
1254        % Check slow convergence
1255        if SlowConvergence
1256            Converged = false;
1257        end
1258    end
1259
1260    if Converged
1261        % Check divergence
1262        if diverging
1263            Converged = false;
1264        end
1265    end
1266
1267    if Converged
1268        % Check previous reject
1269        if PreviousReject
1270            Converged = false;
1271        end
1272    end
1273
1274    if Converged
1275        % Check first step
1276        if FirstStep
1277            Converged = false;
1278        end
1279    end
1280
1281    if Converged
1282        % Check slow convergence
1283        if SlowConvergence
1284            Converged = false;
1285        end
1286    end
1287
1288    if Converged
1289        % Check divergence
1290        if diverging
1291            Converged = false;
1292        end
1293    end
1294
1295    if Converged
1296        % Check previous reject
1297        if PreviousReject
1298            Converged = false;
1299        end
1300    end
1301
1302    if Converged
1303        % Check first step
1304        if FirstStep
1305            Converged = false;
1306        end
1307    end
1308
1309    if Converged
1310        % Check slow convergence
1311        if SlowConvergence
1312            Converged = false;
1313        end
1314    end
1315
1316    if Converged
1317        % Check divergence
1318        if diverging
1319            Converged = false;
1320        end
1321    end
1322
1323    if Converged
1324        % Check previous reject
1325        if PreviousReject
1326            Converged = false;
1327        end
1328    end
1329
1330    if Converged
1331        % Check first step
1332        if FirstStep
1333            Converged = false;
1334        end
1335    end
1336
1337    if Converged
1338        % Check slow convergence
1339        if SlowConvergence
1340            Converged = false;
1341        end
1342    end
1343
1344    if Converged
1345        % Check divergence
1346        if diverging
1347            Converged = false;
1348        end
1349    end
1350
1351    if Converged
1352        % Check previous reject
1353        if PreviousReject
1354            Converged = false;
1355        end
1356    end
1357
1358    if Converged
1359        % Check first step
1360        if FirstStep
1361            Converged = false;
1362        end
1363    end
1364
1365    if Converged
1366        % Check slow convergence
1367        if SlowConvergence
1368            Converged = false;
1369        end
1370    end
1371
1372    if Converged
1373        % Check divergence
1374        if diverging
1375            Converged = false;
1376        end
1377    end
1378
1379    if Converged
1380        % Check previous reject
1381        if PreviousReject
1382            Converged = false;
1383        end
1384    end
1385
1386    if Converged
1387        % Check first step
1388        if FirstStep
1389            Converged = false;
1390        end
1391    end
1392
1393    if Converged
1394        % Check slow convergence
1395        if SlowConvergence
1396            Converged = false;
1397        end
1398    end
1399
1400    if Converged
1401        % Check divergence
1402        if diverging
1403            Converged = false;
1404        end
1405    end
1406
1407    if Converged
1408        % Check previous reject
1409        if PreviousReject
1410            Converged = false;
1411        end
1412    end
1413
1414    if Converged
1415        % Check first step
1416        if FirstStep
1417            Converged = false;
1418        end
1419    end
1420
1421    if Converged
1422        % Check slow convergence
1423        if SlowConvergence
1424            Converged = false;
1425        end
1426    end
1427
1428    if Converged
1429        % Check divergence
1430        if diverging
1431            Converged = false;
1432        end
1433    end
1434
1435    if Converged
1436        % Check previous reject
1437        if PreviousReject
1438            Converged = false;
1439        end
1440    end
1441
1442    if Converged
1443        % Check first step
1444        if FirstStep
1445            Converged = false;
1446        end
1447    end
1448
1449    if Converged
1450        % Check slow convergence
1451        if SlowConvergence
1452            Converged = false;
1453        end
1454    end
1455
1456    if Converged
1457        % Check divergence
1458        if diverging
1459            Converged = false;
1460        end
1461    end
1462
1463    if Converged
1464        % Check previous reject
1465        if PreviousReject
1466            Converged = false;
1467        end
1468    end
1469
1470    if Converged
1471        % Check first step
1472        if FirstStep
1473            Converged = false;
1474        end
1475    end
1476
1477    if Converged
1478        % Check slow convergence
1479        if SlowConvergence
1480            Converged = false;
1481        end
1482    end
1483
1484    if Converged
1485        % Check divergence
1486        if diverging
1487            Converged = false;
1488        end
1489    end
1490
1491    if Converged
1492        % Check previous reject
1493        if PreviousReject
1494            Converged = false;
1495        end
1496    end
1497
1498    if Converged
1499        % Check first step
1500        if FirstStep
1501            Converged = false;
1502        end
1503    end
1504
1505    if Converged
1506        % Check slow convergence
1507        if SlowConvergence
1508            Converged = false;
1509        end
1510    end
1511
1512    if Converged
1513        % Check divergence
1514        if diverging
1515            Converged = false;
1516        end
1517    end
1518
1519    if Converged
1520        % Check previous reject
1521        if PreviousReject
1522            Converged = false;
1523        end
1524    end
1525
1526    if Converged
1527        % Check first step
1528        if FirstStep
1529            Converged = false;
1530        end
1531    end
1532
1533    if Converged
1534        % Check slow convergence
1535        if SlowConvergence
1536            Converged = false;
1537        end
1538    end
1539
1540    if Converged
1541        % Check divergence
1542        if diverging
1543            Converged = false;
1544        end
1545    end
1546
1547    if Converged
1548        % Check previous reject
1549        if PreviousReject
1550            Converged = false;
1551        end
1552    end
1553
1554    if Converged
1555        % Check first step
1556        if FirstStep
1557            Converged = false;
1558        end
1559    end
1560
1561    if Converged
1562        % Check slow convergence
1563        if SlowConvergence
1564            Converged = false;
1565        end
1566    end
1567
1568    if Converged
1569        % Check divergence
1570        if diverging
1571            Converged = false;
1572        end
1573    end
1574
1575    if Converged
1576        % Check previous reject
1577        if PreviousReject
1578            Converged = false;
1579        end
1580    end
1581
1582    if Converged
1583        % Check first step
1584        if FirstStep
1585            Converged = false;
1586        end
1587    end
1588
1589    if Converged
1590        % Check slow convergence
1591        if SlowConvergence
1592            Converged = false;
1593        end
1594    end
1595
1596    if Converged
1597        % Check divergence
1598       
```

```

115      info.nFun = info.nFun+1;
116      R = G - hgamm*aF(:, i) - phi;
117
118      rNewton = norm(R./(absTol + abs(G).*relTol), inf);
119      Converged = (rNewton < tau);
120
121      % Newton Iterations
122      while Converged && diverging && SlowConvergence
123          iter(i) = iter(i)+1;
124          dX = U\(\L\)(R(pivot,1)); % LU factorize residual term
125          info.nBack = info.nBack+1;
126          X = X - dX;
127          rNewtonOld = rNewton;
128          [F(:, i), G] = feval(fun, T, X, varargin{:});
129          info.nFun = info.nFun+1;
130
131          R = G - hgamm*aF(:, i) - phi; % Update residual term
132          rNewton = norm(R./(absTol + abs(G).*relTol), inf);
133          alpha = max(alpha, rNewton/rNewtonOld);
134
135          Converged = (rNewton < tau);
136          diverging = (alpha >= 1);
137          SlowConvergence = (iter(i) >= itermax);
138
139      end
140      diverging = (alpha >= 1)*i; % check for divergence
141  end
142
143  % Store variables for output
144  nstep = info.nStep;
145  stats.t(nstep) = t;
146  stats.h(nstep) = h;
147  stats.r(nstep) = NaN;
148  stats.iter(nstep, :) = iter;
149  stats.Converged(nstep) = Converged;
150  stats.Diverged(nstep) = diverging;
151  stats.AcceptStep(nstep) = false;
152
153  % Check which stage has slow convergence (reaches maximum nr of
154  % iterations)
155  stats.SlowConv(nstep) = SlowConvergence*i;
156  iter(:) = 0;
157
158  % Error and Convergence Controller
159  if Converged
160      % Error estimation
161      e = F*(h*d);
162      r = norm(e./(absTol + abs(G).*relTol), inf);
163      AcceptCurrentStep = (r<=1.0);
164      r = max(r, eps);
165      stats.r(nstep) = r;
166      % Step Length Controller
167      if AcceptCurrentStep
168          stats.AcceptStep(nstep) = true;
169          info.nAccept = info.nAccept+1;

```

```

169      if FirstStep || PreviousReject || ConvergenceRestriction
170          % Aymptotic step size controller
171          hr = 0.75*(epsilon/r)^ke0;
172      else
173          % Predictive controller
174          s0 = (h/hacc);
175          s1 = max(hrmin,min(hrmax,(racc/r)^ke1));
176          s2 = max(hrmin,min(hrmax,(epsilon/r)^ke2));
177          hr = 0.95*s0*s1*s2;
178      end
179      racc = r;
180      hacc = h;
181      FirstStep = false;
182      PreviousReject = false;
183      ConvergenceRestriction = false;
184
185      % Next Step
186      t = T;
187      x = X;
188      g = G;
189      F(:,1) = F(:,s);
190
191  else % Reject current step
192      info.nFail = info.nFail+1;
193      if PreviousReject
194          kest = log(r/rrej)/(log(h/hrej));
195          kest = min(max(0.1,kest),phat);
196          hr = max(hrmin,min(hrmax,((epsilon/r)^(1/kest))));
197      else
198          hr = max(hrmin,min(hrmax,((epsilon/r)^ke0)));
199      end
200      rrej = r;
201      hrej = h;
202      PreviousReject = true;
203  end
204
205  % Convergence control
206  halpha = (alpharef/alpha);
207  if (alpha > alpharef)
208      ConvergenceRestriction = true;
209      if hr < halpha
210          h = max(hrmin,min(hrmax,hr))*h;
211      else
212          h = max(hrmin,min(hrmax,halpha))*h;
213      end
214  else
215      h = max(hrmin,min(hrmax,hr))*h;
216  end
217  h = max(1e-8,h);
218  if (t+h) > tf
219      h = tf-t;
220  end
221
222  % Jacobian Update Strategy
223  FreshJacobian = false;

```

```

224      if alpha > alphaJac
225          [dfdx,dgdx] = feval(jac,t,x,varargin{:});
226          info.nJac = info.nJac+1;
227          FreshJacobian = true;
228          hgamma = h*gamma;
229          dRdx = dgdx - hgamma*dfdx;
230          [L,U,pivot] = lu(dRdx, 'vector');
231          info.nLU = info.nLU+1;
232          hLU = h;
233      elseif (abs(h-hLU)/hLU) > alphaLU
234          hgama = h*gamma;
235          dRdx = dgdx-hgamma*dfdx;
236          [L,U,pivot] = lu(dRdx, 'vector');
237          info.nLU = info.nLU+1;
238          hLU = h;
239      end
240  else % not converged
241      info.nFail=info.nFail+1;
242      AcceptCurrentStep = false;
243      ConvergenceRestriction = true;
244      if FreshJacobian && diverging
245          h = max(0.5*hrmin,alpharef/alpha)*h;
246          info.nDiverge = info.nDiverge+1;
247      elseif FreshJacobian
248          if alpha > alpharef
249              h = max(0.5*hrmin,alpharef/alpha)*h;
250          else
251              h = 0.5*h;
252          end
253      end
254      if FreshJacobian
255          [dfdx,dgdx] = feval(jac,t,x,varargin{:});
256          info.nJac = info.nJac+1;
257          FreshJacobian = true;
258      end
259      hgama = h*gamma;
260      dRdx = dgdx - hgama*dfdx;
261      [L,U,pivot] = lu(dRdx, 'vector');
262      info.nLU = info.nLU+1;
263      hLU = h;
264  end
265
266
267  % Store variables for output
268  if AcceptCurrentStep
269      nAccept = info.nAccept;
270      if nAccept > length(Tout)
271          Tout = [Tout; zeros(Nsize,1)];
272          Xout = [Xout; zeros(Nsize,nx)];
273          Gout = [Gout; zeros(Nsize,nx)];
274      end
275      Tout(nAccept,1) = t;
276      Xout(nAccept,:) = x.';
277      Gout(nAccept,:) = g.';
278  end

```

```

279 end
280 info.nSlowConv = length(find(stats.SlowConv));
281 nAccept = info.nAccept;
282 Tout = Tout(1:nAccept,:);
283 Xout = Xout(1:nAccept,:,:);
284 Gout = Gout(1:nAccept,:,:);

```

Listing A.15: The ESDIRK23 method with adaptive step size.

A.2 Drivers

A.2.1 Problem 1 Driver

```

1 %% Problem 1
2 close all; clear all; clc;
3 %1. Provide the analytical solution to the test equation
4
5 lambda = -1;
6 x0=1;
7 t = linspace(0,20,100);
8
9 % Plot the analytical solution
10 plot(t, exp(x0.*lambda.*t));
11
12 %% Exercise 1.3: Local and global truncation errors for the test eq.
13 %% Exercise 1.3a) Explicit Euler method (fixed step size)
14 close all; clear all; clc
15
16 lambda = -1;
17 x0=1;
18
19 t0=0;
20 tN=10;
21 N=50;
22 h = (tN-t0)/N;
23
24 [T,X] = ExplicitEulerFixedStepSize(@TestEq,t0,tN,N,x0,lambda);
25
26 % Show analytical solution vs the approximations
27 plot(T, exp(x0.*lambda.*T)), hold on,
28 plot(T,X);
29 xlabel('t')
30 ylabel('x(t)')
31 title('Test Equation')
32 legend('Analytical Solution','Explicit Euler Fixed Step Size')
33
34 % Error estimation
35
36 % Global error:
37 e_glob = X-exp(x0.*lambda.*T);

```

```
38 %Local error:
39 e_loc = (1 + lambda*h).*X- exp(lambda*h).*X;
40
41 fprintf('The Global error is %.4f\n',norm(e_glob))
42 fprintf('The local error is %.4f\n',norm(e_loc))
43
44 %% Exercise 1.3 b) The implicit Euler method (fixed step size)
45 clear all;
46
47 lambda = -1;
48 x0=1;
49
50 ta=0;
51 tb=10;
52 N=100;
53 h = (tb-ta)/N;
54
55 [T,X] = ImplicitEulerFixedStepSize2D(@TestEqJac,ta,tb,N,x0,lambda);
56
57 figure,
58 plot(T, exp(x0.*lambda.*T)), hold on
59 plot(T,X)
60 xlabel('t'), ylabel('x(t)')
61 title('Test Equation')
62 legend('Analytical Solution','Implicit Euler Fixed Step Size')
63
64
65 %% Global error:
66 e_glob = X-exp(x0.*lambda.*T);
67
68 %% Local error
69 e_loc = X.*((1/(1-lambda*h))- exp(lambda*h).*X);
70
71 fprintf('The Global error is %.4f\n',norm(e_glob))
72 fprintf('The local error is %.4f\n',norm(e_loc))
73
74 %% Exercise 1.3c) The classical Runge-Kutta method
75 clear all;
76
77 lambda = -1;
78 x0=1;
79
80 ta=0;
81 tb=10;
82 N=100;
83
84 tspan = [ta,tb];
85
86 h=(tb-ta)/N;
87
88
89 solver.stages = 4; % Number of stages in ERK method
90 solver.AT = [0,0,0,0;0.5,0,0,0;0,0.5,0,0;0,0,1,0]'; % Transpose of A-matrix
91 in Butcher tableau
```

```

92 solver.b = [1/6,1/3,1/3,1/6]'; % b-vector in Butcher tableau
93 solver.c = [0;1/2;1/2;1]; % c-vector in Butcher tableau
94
95 [T,X]=ExplicitRungeKuttaSolver(@TestEq,tspan,x0,h,solver,lambda);
96
97
98 plot(T, exp(x0.*lambda.*T));
99 hold on
100 plot(T,X);
101 xlabel('t')
102 ylabel('x(t)')
103 title('Test Equation')
104 legend('Analytical Solution','Explicit Euler Fixed Step Size')
105
106
107 %Global error:
108 e_glob = X-exp(x0.*lambda.*T);
109
110 %Local error:
111 for i=1:length(T)-1
112
113 e_loc(i) = X(i+1) - exp(lambda*h)*X(i);
114
115 end
116
117 format long
118
119 fprintf('The Global error is %.10f\n',norm(e_glob))
120 fprintf('The local error is %.10f\n',norm(e_loc))
121
122 %% Exercise 1.4. Plot local error vs the time step for the numerical methods
123 clear all
124
125 lambda = -1;
126 x0=1;
127
128 ta=0;
129 tb=1;
130 N=100;
131
132
133 solver.stages = 4; % Number of stages in ERK method
134 solver.AT = [0,0,0,0;0.5,0,0,0;0,0.5,0,0;0,0,1,0]'; % Transpose of A-matrix
135 % in Butcher tableau
136 solver.b = [1/6,1/3,1/3,1/6]'; % b-vector in Butcher tableau
137 solver.c = [0;1/2;1/2;1]; % c-vector in Butcher tableau
138 H = logspace(-3,0,20);
139
140 e_loc_runge=[];
141 e_loc_EE=[];
142 e_loc_IE=[];
143
144 for i=1:length(H)
145

```

```

146 [T,X_runge] = ExplicitRungeKuttaSolver(@TestEq,[0 H(i)],x0,H(i),solver,
147     lambda);
148 e_loc_runge(i)= abs(X_runge(end)-X_runge(end-1)*exp(H(i)*lambda));
149 [T,X_EE] = ExplicitEulerFixedStepSize(@TestEq,0,H(i),1,x0,lambda);
150 e_loc_EE(i)= abs(X_EE(end)- exp(lambda*H(i))*X_EE(end-1));
151 [T,X_IE] = ImplicitEulerFixedStepSize2D(@TestEqJac,0,H(i),1,x0,lambda);
152 e_loc_IE(i)= abs(X_IE(end)- exp(lambda*H(i))*X_IE(end-1));
153
154 end
155
156 figure, h = gcf;
157 loglog(H,e_loc_EE,'r-o','LineWidth',1)
158 hold on
159 loglog(H,e_loc_IE,'b-o','LineWidth',1)
160 hold on
161 loglog(H,e_loc_runge,'m-o','LineWidth',1)
162
163 ylabel('|e|', 'FontSize', 24)
164 xlabel('h', 'FontSize', 24)
165 title('Local Error', 'FontSize', 28)
166 legend('Explicit Euler', 'Implicit Euler', 'Classical Runge Kutta',...
167     'Location', 'NorthWest', 'fontsize', 16)
168 set(gca,'fontsize',18)
169
170 fprintf('The norm of the local error for the %s method is %.10f\n',...
171     'Explicit Euler',norm(e_loc_EE))
172 fprintf('The norm of the local error for the %s method is %.10f\n',...
173     'Implicit Euler',norm(e_loc_IE))
174 fprintf('The norm of the local error for the %s method is %.10f\n',...
175     'Classical Runge Kutta',norm(e_loc_runge))
176
177
178 fit1 = polyfit(log(H),log(e_loc_EE),1);
179 fprintf('The order of %s method is: %i\n', 'the Explicit Euler',...
180     round(fit1(1))-1)
181
182 %For the calculation of the order of Implicit Euler I only take the first
183 %15 points, as from the plot it can be seen that the slope goes downwards
184 %just in the end and if fitting with all the points it says the order is 0.
185 fit2 = polyfit(log(H),log(e_loc_IE),1);
186 fprintf('The order of %s method is: %i\n', 'the Implicit Euler',...
187     round(fit2(1))-1)
188
189 fit3 = polyfit(log(H([2:length(e_loc_runge)])),...
190     log(e_loc_runge([2:length(e_loc_runge)])),1);
191 fprintf('The order of %s method is: %i\n', 'the Classical Runge Kutta',...
192     round(fit3(1))-1)
193
194 %% Exercise 1.5. Plot the global error at t=1.0 for the three methods.
195 clear all
196
197 lambda = -1;
198 x0=1;
199

```

```

200
201 ta=0;
202 tb=1;
203 N=100;
204
205 solver.stages = 4; % Number of stages in ERK method
206 solver.AT = [0,0,0,0;0.5,0,0,0;0,0.5,0,0;0,0,1,0]'; % Transpose of A-matrix
207 % in Butcher tableau
208 solver.b = [1/6,1/3,1/3,1/6]'; % b-vector in Butcher tableau
209 solver.c = [0;1/2;1/2;1]; % c-vector in Butcher tableau
210
211
212 H = logspace(-3,0,20);
213
214 e_glob_runge=[];
215 e_glob_EE=[];
216 e_gob_IE=[];
217
218 Ns=linspace(10,1000,10);
219
220 for i=1:length(Ns)
221
222 N=Ns(i);
223 h=1/N;
224
225 [T,X_runge] = ExplicitRungeKuttaSolver(@TestEq,[ta tb],x0,h,solver,
226 lambda);
226 e_glob_runge(i)= abs(X_runge(end)-exp(lambda.*T(end))*x0);
227
228 [T,X_EE] = ExplicitEulerFixedStepSize(@TestEq,0,tb,N,x0,lambda);
229 e_glob_EE(i)= abs(X_EE(end)-exp(lambda.*T(end))*x0);
230
231 [T,X_IE] = ImplicitEulerFixedStepSize2D(@TestEqJac,0,tb,N,x0,lambda);
232 e_glob_IE(i)= abs(X_IE(end)-exp(lambda.*T(end))*x0);
233
234
235 end
236
237
238 H=1./Ns;
239
240 figure, h2=gcf;
241 loglog(H,e_glob_EE,'r-o','LineWidth',1)
242 hold on
243 loglog(H,e_glob_IE,'b-o','LineWidth',1)
244 hold on
245 loglog(H,e_glob_runge,'m-o','LineWidth',1)
246
247 ylabel('|e|', 'FontSize', 24)
248 xlabel('h', 'FontSize', 24)
249 title('Global Error', 'FontSize', 28)
250 legend('Explicit Euler','Implicit Euler','Classical Runge Kutta',...
251 'Location', 'NorthWest','fontsize',16)
252 set(gca,'fontsize',18)
253

```

```

254
255
256 fit1 = polyfit(log(H),log(e_glob_EE),1);
257 fit2 = polyfit(log(H),log(e_glob_IE),1);
258 fit3 = polyfit(log(H),log(e_glob_runge),1);
259
260
261 fprintf('The norm of the global error for the %s method is %.10f\n', '
262     Explicit Euler',norm(e_glob_EE))
263 fprintf('The norm of the global error for the %s method is %.10f\n', '
264     Implicit Euler',norm(e_glob_IE))
265 fprintf('The norm of the global error for the %s method is %.10f\n\n', '
266     Classical Runge Kutta',norm(e_glob_runge))
267 fprintf('The order of the global error for %s is: %i\n', 'the Explicit Euler'
268     ,round(fit1(1)))
269 fprintf('The order of the global error for %s is: %i\n', 'the Implicit Euler'
270     ,round(fit2(1)))
271 fprintf('The order of the global error for %s is: %i\n', 'the Classical Runge
272     Kutta',round(fit3(1)))
273
274 %% Exercise 1.6 Plot the stability regions for the three methods
275 close all; clear all; clc
276
277 alpha = -5:0.01:5;
278 beta = -5:0.01:5;
279
280 nreal = length(alpha);
281 nimag = length(beta);
282
283 % For classical Runge Kutta
284 solver.stages = 4; % Number of stages
285 % Butcher tableau components
286 solver.AT = [0,0,0,0;0.5,0,0,0;0,0.5,0,0;0,0,1,0]'; % Transpose of A-matrix
287 solver.b = [1/6,1/3,1/3,1/6]';
288 solver.c = [0;1/2;1/2;1];
289
290 A = solver.AT';
291 b = solver.b;
292 c = solver.c;
293
294 I = eye(size(A));
295 e = ones(size(A,1),1);
296
297 % Compute stability regions for the three methods
298 for kreal = 1:nreal
299     for kimag = 1:nimag
300
301         z = alpha(kreal) + i*beta(kimag);
302
303         % Compute transfer function for classical Runge-Kutta
304         tmp = (I-z*A)\e;
305         R_runge = 1 + z*b'*tmp;
306         absR_runge(kimag,kreal) = abs(R_runge);
307
308         % Compute transfer function for explicit and implicit Euler

```

```

303 R_EE = 1+z ;
304 R_IE = 1/(1-z);
305 absR_EE(kimag , kreal) = abs(R_EE);
306 absR_IE(kimag , kreal) = abs(R_IE);
307 end
308 end
309
310 figure , p = gcf;
311 fs = 19; % set fontsize for labels and title
312 imagesc(alpha ,beta ,absR_runge,[0 1]);
313 grid on
314 colorbar
315 axis image xy
316 xticks(linspace(-5,5,11)), xtickangle(0),
317 yticks(linspace(-5,5,11)), ytickangle(0),
318 xlabel('Re(z)', 'fontsize',fs), ylabel('Im(z)', 'fontsize',fs)
319 title('Classical Runge-Kutta, |R(z)|', 'fontsize',fs+5)
320 set(gca,'fontsize',16)
321
322
323 figure ; p2 = gcf;
324 imagesc(alpha ,beta ,absR_EE,[0 1]);
325 grid on
326 colorbar
327 axis image xy
328 xticks(linspace(-5,5,11)), xtickangle(0),
329 yticks(linspace(-5,5,11)), ytickangle(0),
330 xlabel('Re(z)', 'fontsize',fs), ylabel('Im(z)', 'fontsize',fs)
331 title('Explicit Euler, |R(z)|', 'fontsize',fs+5)
332 set(gca,'fontsize',16)
333
334
335 figure ,
336 p3 = gcf;
337 imagesc(alpha ,beta ,absR_IE,[0 1]);
338 grid on
339 colorbar
340 axis image xy
341 xticks(linspace(-5,5,11)), xtickangle(0),
342 yticks(linspace(-5,5,11)), ytickangle(0),
343 xlabel('Re(z)', 'fontsize',fs), ylabel('Im(z)', 'fontsize',fs)
344 title('Implicit Euler, |R(z)|', 'fontsize',fs+5)
345 set(gca,'fontsize',16)
346
347
348 %% Exercise 1.6 cont: Check stability of the three methods
349 % A- and L-stability: Lecture 8B - slide 24
350
351 lis = 1:nreal;
352 ind_ltzero=lis(alpha<0);
353
354 EE_Astable = absR_EE(:,ind_ltzero);
355 max_EE=max(max(EE_Astable));
356
357 IE_Astable = absR_IE(:,ind_ltzero);

```

```

358 max_IE=max(max(IE_Astable));
359
360 runge_Astable = absR_runge(:, ind_ltzero);
361 max_runge = max(max(runge_Astable));
362
363
364 fprintf(...);
365 'The highest value for Re(z)<0 in the stability region for Explicit Euler is
366 : %f and thus it is not A-stable\n',max_EE)
366 fprintf(...);
367 'The highest value for Re(z)<0 in the stability region for Implicit Euler is
368 : %f and thus it is A-stable\n',max_IE)
368 fprintf(...);
369 'The highest value for Re(z)<0 in the stability region for Classical Runge
370 Kutta is: %f and thus it is not A-stable\n\n',max_runge)
371
372 %Which of the methods are L-stable?
373
374 %Lets check some low values
375 z=-10000-i*10000;
376
377 R_EE=1+z;
378 R_IE=1/(1-z);
379 absR_EE=abs(R_EE);
380 absR_IE=abs(R_IE);
381
382
383 tmp=(I-z*A)\e;
384 R_runge=1+z*b'*tmp;
385 absR_runge=abs(R_runge);
386
387 fprintf('The stability region for Explicit Euler when z->inf is: %f and thus
388 it is not L-stable\n',absR_EE)
388 fprintf('The stability region for Implicit Euler when z->inf is: %f and thus
389 it is L-stable\n',absR_IE)
389 fprintf('The stability region for Classical Runge Kutta when z->inf is: %f
and thus it is not L-stable\n',absR_runge)

```

Listing A.16: Driver for problem 1 on the test equation.

A.2.2 Problem 2 Driver

```

1 %% Problem 2: Explicit ODE Solver
2 %
3 % -----
4 % Exercise 2.2-4: Explicit Euler Fixed Step Size
5 close all; clear all; clc;
6
7
8 % Van der Pol problem parameters

```

```

9 x0 = [1.0 ,1.0]';
10 mu = [3 , 20];
11 N = 10000; % Change this to increase/decrease step size
12 tspan = [0 ,50];
13
14
15
16 for N = [3000 ,10000]
17
18 [T,X] = ExplicitEulerFixedStepSize(@VanderPolfunjac ,tspan(1) ,tspan(2) ,N,
19 x0,mu(1));
20 [T2,X2] = ExplicitEulerFixedStepSize(@VanderPolfunjac ,tspan(1) ,tspan(2) ,
21 N,x0,mu(2));
22
23
24
25 % Plot results in two figures for each mu
26 fs = 21; % fontsize for labels
27 figure; f = gcf;
28 ax=subplot(3,1,1);
29 plot(T,X(:,1) , 'r' , 'LineWidth' ,1.5);
30 xlabel('t' , 'FontSize' ,fs),ylabel('x_1(t)' , 'FontSize' ,fs),
31 ylim([-3 3]),xlim([0 50]), set(gca , 'fontsize' ,fs -3)
32 subtitle('Van der Pol:  $\mu=3$ ' , 'fontsize' ,fs -2 , 'fontWeight' , 'bold')
33 %title('Explicit Euler Fixed Step Size')
34 ax2=subplot(3,1,2);
35 plot(T,X(:,2) , 'r' , 'LineWidth' ,1.5);
36 xlabel('t' , 'FontSize' ,fs),ylabel('x_2(t)' , 'FontSize' ,fs),
37 ylim([-6 6]),xlim([0 50]), set(gca , 'fontsize' ,fs -3)
38 ax3=subplot(3,1,3);
39 plot(X(:,1),X(:,2) , 'r' , 'LineWidth' ,1.5),
40 ylabel('x_2(t)' , 'FontSize' ,fs), xlabel('x_1(t)' , 'FontSize' ,fs),ylim([-8
41 8])
42 %set(gcf , 'Position' ,[100 100 550 550])
43 %sgtitle({' Explicit Euler - Fixed Step '}, 'FontSize' , fs+4)
44
45 figure; f2 = gcf;
46 ax4=subplot(3,1,1);
47 plot(T2,X2(:,1) , 'r' , 'LineWidth' ,1.5);
48 xlabel('t' , 'FontSize' ,fs),ylabel('x_1(t)' , 'FontSize' ,fs),
49 ylim([-3 3]),xlim([0 50]), set(gca , 'fontsize' ,fs -3)
50 subtitle('Van der Pol:  $\mu=20$ ' , 'fontsize' ,fs -2 , 'fontWeight' , 'bold')
51 ax5=subplot(3,1,2);
52 plot(T2,X2(:,2) , 'r' , 'LineWidth' ,1.5),
53 xlabel('t' , 'FontSize' ,fs),ylabel('x_2(t)' , 'FontSize' ,fs),
54 ylim([-30 30]),xlim([0 50]), set(gca , 'fontsize' ,fs -3)
55 ax6=subplot(3,1,3);
56 plot(X2(:,1),X2(:,2) , 'r' , 'LineWidth' ,1.5),ylim([-33 33]),
57 ylabel('x_2(t)' , 'FontSize' ,fs), xlabel('x_1(t)' , 'FontSize' ,fs)
58 sgtitle({' Explicit Euler - Fixed Step '}, 'FontSize' , fs+4)
59 set(gca , 'fontsize' ,fs -3)
60 set([ax,ax2,ax3,ax4,ax5,ax6] , 'fontsize' ,fs -3)
61
62 end
63
64
65
66
67 %

```

```

61 %% Exercise 2.3-4 Explicit Euler Adaptive Time Step and Error Estimation
62 close all; clear all; clc;
63
64 x0 = [1.0 ,1.0]';
65 mu = [3, 20];
66 abstol = 1e-5;
67 reltol = 1e-5;
68 t0 = 0;
69 fs = 21;
70 tspan = [0 ,50];
71
72 h0 = 0.001; % h0 = 10^-2;
73
74 [T,X,stat] = ExplicitEulerAdaptiveTimeStep(@VanderPolfunjac ,...
75 tspan,x0,h0,abstol,reltol,mu(1));
76 [T2,X2,stat2] = ExplicitEulerAdaptiveTimeStep(@VanderPolfunjac ,...
77 tspan,x0,h0,abstol,reltol,mu(2));
78
79
80 % Plot results in two figures for each mu
81 figure; f3 = gcf;
82 ax = subplot(3,1,1);
83 plot(T,X(:,1), 'r', 'LineWidth',1.5);
84 xlabel('t', 'FontSize',fs), ylabel('x_1(t)', 'FontSize',fs),
85 ylim([-3 3]), xlim([0 50])
86 subtitle({{'Van Der Pol:  $\mu=3$ '}, 'fontsize',fs-2, 'fontweight','bold'})
87 %set(gca, 'fontsize',fs-3)
88 ax2 = subplot(3,1,2);
89 plot(T,X(:,2), 'r', 'LineWidth',1.5);
90 xlabel('t', 'FontSize',fs), ylabel('x_2(t)', 'FontSize',fs),
91 ylim([-6 6]), xlim([0 50])
92 ax3 = subplot(3,1,3);
93 plot(X(:,1),X(:,2), 'r', 'LineWidth',1.5), ylim([-8 8]),
94 ylabel('x_2(t)', 'FontSize',fs), xlabel('x_1(t)', 'FontSize',fs)
95 sgttitle({{'Explicit Euler - Adaptive Step'}}, 'FontSize', fs+4)
96
97
98 figure; f4=gcf;
99 ax4=subplot(3,1,1);
100 line3 = plot(T2,X2(:,1), 'r', 'LineWidth',1.5); ylabel('x_1(t)', 'FontSize',fs)
101 , xlabel('t', 'FontSize',fs), ylim([-3 3]), xlim([0 50])
102 subtitle({{'Van Der Pol:  $\mu=20$ '}, 'fontsize',fs-2, 'fontweight','bold'})
103 ax5 =subplot(3,1,2);
104 plot(T2,X2(:,2), 'r', 'LineWidth',1.5), ylabel('x_2(t)', 'FontSize',fs),
105 xlabel('t', 'FontSize',fs), ylim([-30 30]), xlim([0 50])
106 ax6 = subplot(3,1,3);
107 plot(X2(:,1),X2(:,2), 'r', 'LineWidth',1.5), ylim([-30 30]),
108 ylabel('x_2(t)', 'FontSize',fs), xlabel('x_1(t)', 'FontSize',fs)
109 sgttitle({{'Explicit Euler - Adaptive Step'}}, 'FontSize', fs+4)
110 set([ax,ax2,ax3,ax4,ax5,ax6], 'fontsize',fs-3)
111
112
113 % Look at error estimation and step sizes
114 figure; f5=gcf;

```

```

115 ax=subplot(2,1,1);
116 plot(linspace(0,50, length(stat.r)), stat.r, 'r', 'LineWidth', 1),
117 xlabel('t', 'fontsize',fs), ylabel('r', 'fontsize',fs), %ylim([-0.1 1])
118 subtitle({'Van Der Pol:  $\mu=3$ '}, 'fontsize',fs-3, 'fontWeight', 'bold')
119 ax2=subplot(2,1,2);
120 plot(linspace(0,50, length(stat.h)), stat.h, 'r', 'LineWidth', 1),
121 xlabel('t'), ylabel('h'), %ylim([-0.01 0.04])
122 sgttitle('Explicit Euler - Adaptive Step', 'fontsize', fs+4)
123
124 figure; f6=gcf;
125 ax3=subplot(2,1,1);
126 plot(linspace(0,50, length(stat2.r)), stat2.r, 'r', 'LineWidth', 1),
127 xlabel('t', 'fontsize',fs), ylabel('r', 'fontsize',fs-2), %ylim([0 1])
128 subtitle({'Van Der Pol:  $\mu=20$ '}, 'fontsize',fs-3, 'fontWeight', 'bold')
129 ax4=subplot(2,1,2);
130 plot(linspace(0,50, length(stat2.h)), stat2.h, 'r', 'LineWidth', 1),
131 xlabel('t', 'fontsize',fs), ylabel('h', 'fontsize',fs), %ylim([-0.01 0.3])
132 sgttitle('Explicit Euler - Adaptive Step', 'fontsize', fs+4)
133 set([ax,ax2,ax3,ax4], 'fontsize',fs-3)
134
135
136 fprintf('Van der Pol Explicit Euler Adaptive Step Size  $\mu=3$ \n')
137 T = table(reltol, stat.nfun, stat.nAccept, stat.nReject, 'VariableNames',
138 ...
139 {'Tolerance', 'N. Func', 'N. Accept', 'N. Reject'});
140 disp(T)
141 fprintf('Van der Pol Explicit Euler Adaptive Step Size  $\mu=20$ \n')
142 T = table(reltol, stat2.nfun, stat2.nAccept, stat2.nReject, 'VariableNames',
143 ...
144 {'Tolerance', 'N. Func', 'N. Accept', 'N. Reject'});
145 disp(T)
146
147 %% Exercise 2.5 Compare with Matlabs ODE solvers using different tolerances
148 %% and step sizes
149 close all; clear all; clc;
150
151 fs = 22;
152 plot_results = true; % Change to not plot figures
153 mu = [3,20];
154 x0 = [1.0;1.0];
155 h0 = 0.001;
156 tspan = [0 50];
157
158 abstol = [1e-3, 1e-5, 1e-7];
159 reltol = abstol;
160
161 n = length(abstol);
162 % Initialize stat variables
163 acc3 = zeros(n,1); acc20 = zeros(n,1);
164 rej3 = zeros(n,1); rej20 = zeros(n,1);
165 nfun3 = zeros(n,1); nfun20 = zeros(n,1);
166 N = 5000;

```

```

167 % Add fixed euler to compare
168 [Tf,Xf] = ExplicitEulerFixedStepSize(@VanderPolfunjac,tspan(1),tspan(2),N,x0
169 ,mu(1));
170 [T2f,X2f] = ExplicitEulerFixedStepSize(@VanderPolfunjac,tspan(1),tspan(2),N,
171 x0,mu(2));
172
173 for i = 1:n
174
175 % Matlab's ode solvers
176 options = odeset("RelTol",reltol(i),"AbsTol",abstol(i));%, 'Stats ','on');
177 [Tmu3,Xmu3, statODE45]=ode45(@VanderPolfunjac,tspan,x0,options,mu(1));
178 [Tmu20,Xmu20, statODE45_2]=ode45(@VanderPolfunjac,tspan,x0,options,mu(2)
179 );
180 [T2mu20,X2mu20,statODE15s]=ode15s(@VanderPolfunjac,tspan,x0,options,mu
181 (2));
182 [T2mu3,X2mu3,statODE15s_2]=ode15s(@VanderPolfunjac,tspan,x0,options,mu
183 (1));
184
185 % Using same initial step size as above for DOPRI54
186 [T,X,stat] = ExplicitEulerAdaptiveTimeStep(...
187 @VanderPolfunjac,tspan,x0,h0,abstol(i),reltol(i),mu(1));
188 [T2,X2,stat2] = ExplicitEulerAdaptiveTimeStep(...
189 @VanderPolfunjac,tspan,x0,h0,abstol(i),reltol(i),mu(2));
190
191 if plot_results
192
193 figure, ax=subplot(2,2,1);
194 plot(Tmu3,Xmu3(:,1), 'r',T,X(:,1), 'g',T2mu3,X2mu3(:,1), 'm-',
195 'LineWidth',1),
196 hold on,
197 plot(Tf,Xf(:,1), 'b:', 'LineWidth',1)
198 ylabel('x_1(t)', 'fontsize',fs), xlabel('t', 'fontsize',fs),
199 xlim([0 50])
200
201 ax2=subplot(2,2,2);
202 plot(Tmu3,Xmu3(:,2), 'r',T,X(:,2), 'g',T2mu3,X2mu3(:,2), 'm-',
203 'LineWidth',1),
204 hold on,
205 plot(Tf,Xf(:,2), 'b:', 'LineWidth',1)
206 ylabel('x_2(t)', 'fontsize',fs), xlabel('t', 'fontsize',fs),
207 xlim([0,50])
208 ax3=subplot(2,2,3:4);
209 plot(Xmu3(:,1),Xmu3(:,2), 'r',X(:,1),X(:,2), 'g',X2mu3(:,1),X2mu3(:,2)
210 , 'm-',
211 'LineWidth',1), hold on
212 plot(Xf(:,1),Xf(:,2), 'b:', 'LineWidth',1),
213 xlim([-4 4]), ylim([-6 6])
214 xlabel('x_1(t)', 'fontsize',fs), ylabel('x_2(t)', 'fontsize',fs)
215 legend('ode45','ExplicitEuler (Adaptive)', 'ode15s',...
216 'ExplicitEuler (Fixed)', 'Location', 'SouthEast', 'fontsize',13.4)
217 sgttitle({ 'Van der Pol:  $\mu = 3$ '}, 'fontsize',fs+5),
218 set([ax,ax2,ax3], 'fontsize',fs-3)
219 subtitle({ 'Tolerance: '+string(abstol(i))}, 'fontsize',15, 'fontweight
220 ', 'bold')

```

```

213
214
215 figure , ax=subplot(2,2,1);
216 plot(Tmu20,Xmu20(:,1), 'r' ,T2,X2(:,1) , 'g' ,T2mu20,X2mu20(:,1) , 'm-' ,
217 'LineWidth',1),
218 hold on,
219 plot(T2f,X2f(:,1) , 'b:' , 'LineWidth',1)
220 ylabel('x_1(t)' , 'fontsize' ,fs) , xlabel('t' , 'fontsize' ,fs) ,
221 xlim([0 50])
222
223 ax2=subplot(2,2,2);
224 plot(Tmu20,Xmu20(:,2) , 'r' ,T2,X2(:,2) , 'g' ,T2mu20,X2mu20(:,2) , 'm-' ,
225 'LineWidth',1),
226 hold on,
227 plot(T2f,X2f(:,2) , 'b:' , 'LineWidth',1)
228 ylabel('x_2(t)' , 'fontsize' ,fs) , xlabel('t' , 'fontsize' ,fs) ,
229 xlim([0 ,50])
230 ax3=subplot(2,2,3:4);
231 plot(Xmu20(:,1) ,Xmu20(:,2) , 'r' ,X2(:,1) ,X2(:,2) , 'g' ,X2mu20(:,1) ,
232 X2mu20(:,2) , 'm-' ,...
233 'LineWidth',1), hold on
234 plot(X2f(:,1) ,X2f(:,2) , 'b:' , 'LineWidth',1)
235 xlim([-3 3])
236 xlabel('x_1(t)' , 'fontsize' ,fs) , ylabel('x_2(t)' , 'fontsize' ,fs)
237 legend('ode45' , 'ExplicitEuler (Adaptive)' , 'ode15s' ,...
238 'ExplicitEuler (Fixed)' , 'Location' , 'SouthEast' , 'fontsize' ,13.4)
239 sgttitle({'Van der Pol:  $\mu = 20$ '} , 'fontsize' ,fs+5),
240 set([ax,ax2,ax3] , 'fontsize' ,fs-3)
241 subtitle({{'Tolerance: '+string(abstol(i))}} , 'fontsize' ,15 , 'fontweight'
242 ' , 'bold')
243
244 end
245
246 % Store variables
247 statode45(:,i) = statODE45; statode45_2(:,i) = statODE45_2;
248 statode15s(:,i) = statODE15s; statode15s_2(:,i) = statODE15s_2;
249 acc3(i) = stat.nAccept; acc20(i) = stat2.nAccept;
250 rej3(i) = stat.nReject; rej20(i) = stat2.nReject;
251 nfun3(i) = stat.nfun; nfun20(i) = stat2.nfun;
252
253 end
254
255 varnames = {'Tolerance' , 'N. Func' , 'N. Accept' , 'N. Reject'};
256 fprintf('Van der Pol Explicit Euler Adaptive Step Size  $\mu=3$ \n')
257 T=table(reltol' , nfun3 , acc3 , rej3 , 'VariableNames' , varnames);
258 disp(T)
259 fprintf('Van der Pol Explicit Euler Adaptive Step Size  $\mu=20$ \n')
260 T=table(reltol' , nfun20 , acc20 , rej20 , 'VariableNames' , varnames);
261 disp(T)
262 fprintf('Van der Pol ODE45  $\mu=3$ \n')
263 T=table(reltol' , statode45(3,:) , statode45(1,:) , statode45(2,:) ,
264 'VariableNames' , varnames);
265 disp(T)
266 fprintf('Van der Pol ODE45  $\mu=20$ \n')

```

```

262 T=table('reltol',statode45_2(3,:)',statode45_2(1,:)',statode45_2(2,:)',...
263     'VariableNames', varnames);
264 disp(T)
265 fprintf('Van der Pol ODE15s mu=3\n')
266 T=table('reltol',statode15s(3,:)',statode15s_2(1,:)',statode15s_2(2,:)',...
267     'VariableNames', varnames);
268 disp(T)
269 fprintf('Van der Pol ODE15s mu=20\n')
270 T=table('reltol',statode15s_2(3,:)',statode15s(1,:)',statode15s(2,:)',...
271     'VariableNames', varnames);
272 disp(T)

```

Listing A.17: Driver for problem 2 on the explicit Euler method.

A.2.3 Problem 3 Driver

```

1 %% Problem 3: Implicit ODE Solver
2 %
3 % -----
4 %>>> Exercise 3.3-4: Explicit Euler Fixed Step Size
5 close all; clear all; clc;
6
7
8 % Van der Pol problem parameters
9 x0 = [1.0,1.0]';
10 mu = [3, 20];
11 tspan = [0,50];
12
13 for N=[3000,10000] % Steps less than 10000 gives a large error for mu=20
14     [T,X] = ImplicitEulerFixedStepSize(@VanderPolfunjac,tspan(1),tspan(2),N,
15         x0,mu(1));
16     [T2,X2] = ImplicitEulerFixedStepSize(@VanderPolfunjac,tspan(1),tspan(2),
17         N,x0,mu(2));
18
19 % Plot results in two figures for each mu
20 fs = 20; % fontsize for labels
21 figure; f = gcf;
22 ax=subplot(3,1,1);
23 plot(T,X(:,1),'r','LineWidth',1.5);
24 xlabel('t','FontSize',fs), ylabel('x_1(t)','FontSize',fs),
25 ylim([-3 3]), xlim([0 50]), set(gca,'fontsize',fs-3)
26 subtitle('Van der Pol: mu=3','fontSize',fs-2,'fontWeight','bold')
27 ax2=subplot(3,1,2);
28 plot(T,X(:,2),'r','LineWidth',1.5);
29 xlabel('t','FontSize',fs), ylabel('x_2(t)','FontSize',fs),
30 ylim([-6 6]), xlim([0 50]), set(gca,'fontSize',fs-3)
31 ax3=subplot(3,1,3);
32 plot(X(:,1),X(:,2),'r','LineWidth',1.5),
33 xlim([-3 3])

```

```

33 ylabel('x_2(t)', 'FontSize', fs), xlabel('x_1(t)', 'FontSize', fs), ylim([-8
34 8])
35 sgttitle({'Implicit Euler - Fixed Step'}, 'FontSize', fs+4)
36
37 figure; f2 = gcf;
38 ax4=subplot(3,1,1);
39 plot(T2,X2(:,1), 'r', 'LineWidth', 1.5);
40 xlabel('t', 'FontSize', fs), ylabel('x_1(t)', 'FontSize', fs),
41 ylim([-3 3]), xlim([0 50]), set(gca, 'fontsize', fs-3)
42 subtitle('Van der Pol: mu=20', 'fontsize', fs-2, 'fontweight', 'bold')
43 ax5=subplot(3,1,2);
44 plot(T2,X2(:,2), 'r', 'LineWidth', 1.5),
45 xlabel('t', 'FontSize', fs), ylabel('x_2(t)', 'FontSize', fs),
46 ylim([-30 30]), xlim([0 50]), set(gca, 'fontsize', fs-3)
47 ax6=subplot(3,1,3);
48 plot(X2(:,1),X2(:,2), 'r', 'LineWidth', 1.5), ylim([-33 33]),
49 xlabel('x_2(t)', 'FontSize', fs), xlabel('x_1(t)', 'FontSize', fs)
50 sgttitle({'Implicit Euler - Fixed Step'}, 'FontSize', fs+4)
51
52 end
53 set([ax,ax2,ax3,ax4,ax5,ax6], 'fontsize', fs-3)
54
55 % -----
56 %% Exercise 3.3-4 Implicit Euler Adaptive Time Step and Error Estimation
57 close all; clear all; clc;
58
59 x0 = [1.0,1.0]';
60 mu = [3, 20];
61 abstol = 1e-5;
62 reltol = 1e-5;
63 t0 = 0;
64 fs = 20;
65 tspan = [0,50];
66
67 h0 = 0.001; % h0 = 10^-2;
68
69
70 [T,X,stat] = ImplicitEulerAdaptiveStep(@VanderPolfunjac, ...
71 tspan,x0,h0,abstol,reltol,mu(1));
72
73
74 [T2,X2,stat2] = ImplicitEulerAdaptiveStep(@VanderPolfunjac, ...
75 tspan,x0,h0,abstol,reltol,mu(2));
76
77
78
79 % Plot results in two figures for each mu
80 figure; f3 = gcf;
81 ax = subplot(3,1,1);
82 plot(T,X(:,1), 'r', 'LineWidth', 1.5);
83 xlabel('t', 'FontSize', fs), ylabel('x_1(t)', 'FontSize', fs),
84 ylim([-3 3]), xlim([0 50])
85 subtitle({'Van Der Pol: mu=3'}, 'fontsize', fs-2, 'fontweight', 'bold')
86 ax2 = subplot(3,1,2);

```

```

87 plot(T,X(:,2), 'r', 'LineWidth', 1.5);
88 xlabel('t', 'FontSize', fs), ylabel('x_2(t)', 'FontSize', fs),
89 %ylim([-6 6]), xlim([0 50])
90 ax3 = subplot(3,1,3);
91 plot(X(:,1),X(:,2), 'r', 'LineWidth', 1.5), ylim([-8 8]),
92 ylabel('x_2(t)', 'FontSize', fs), xlabel('x_1(t)', 'FontSize', fs)
93 sgtitle({'Implicit Euler - Adaptive Step'}, 'FontSize', fs+4)
94
95
96 figure; f4=gcf;
97 ax4=subplot(3,1,1);
98 line3 = plot(T2,X2(:,1), 'r', 'LineWidth', 1.5); ylabel('x_1(t)', 'FontSize', fs)
99 xlabel('t', 'FontSize', fs), ylim([-3 3]), xlim([0 50])
100 subtitle({'Van Der Pol:  $\mu=20$ '}, 'fontsize', fs-2, 'fontweight', 'bold')
101 ax5 = subplot(3,1,2);
102 plot(T2,X2(:,2), 'r', 'LineWidth', 1.5), ylabel('x_2(t)', 'FontSize', fs),
103 xlabel('t', 'FontSize', fs), ylim([-30 30]), xlim([0 50])
104 ax6 = subplot(3,1,3);
105 plot(X2(:,1),X2(:,2), 'r', 'LineWidth', 1.5), ylim([-30 30]),
106 ylabel('x_2(t)', 'FontSize', fs), xlabel('x_1(t)', 'FontSize', fs)
107 sgtitle({'Implicit Euler - Adaptive Step'}, 'FontSize', fs+4)
108
109
110 % Look at error estimation and step sizes
111 figure; f5=gcf;
112 ax=subplot(2,1,1);
113 plot(linspace(0,50, length(stat.r)), stat.r, 'r', 'LineWidth', 1),
114 xlabel('t', 'fontsize', fs), ylabel('r', 'fontsize', fs), %ylim([-0.1 1])
115 subtitle({'Van Der Pol:  $\mu=3$ '}, 'fontsize', fs-3, 'fontweight', 'bold')
116 ax2=subplot(2,1,2);
117 plot(linspace(0,50, length(stat.h)), stat.h, 'r', 'LineWidth', 1),
118 xlabel('t', 'FontSize', fs), ylabel('h', 'FontSize', fs), ylim([0 0.04])
119 sgtitle('Implicit Euler - Adaptive Step', 'fontsize', fs+4)
120
121 figure; f6=gcf;
122 ax3=subplot(2,1,1);
123 plot(linspace(0,50, length(stat2.r)), stat2.r, 'r', 'LineWidth', 1),
124 xlabel('t', 'fontsize', fs), ylabel('r', 'fontsize', fs-2), %ylim([0 1])
125 subtitle({'Van Der Pol:  $\mu=20$ '}, 'fontsize', fs-3, 'fontweight', 'bold')
126 ax4=subplot(2,1,2);
127 plot(linspace(0,50, length(stat2.h)), stat2.h, 'r', 'LineWidth', 1),
128 xlabel('t', 'fontsize', fs), ylabel('h', 'fontsize', fs), %ylim([-0.01 0.3])
129 sgtitle('Implicit Euler - Adaptive Step', 'fontsize', fs+4)
130
131 set([ax,ax2,ax3,ax4], 'fontsize', fs-3)
132
133 fprintf('Van der Pol Explicit Euler Adaptive Step Size  $\mu=3\n$ ')
134 T = table(reltol, stat.nfun, stat.nAccept, stat.nReject, 'VariableNames',
135 ...
136 {'Tolerance', 'N. Func', 'N. Accept', 'N. Reject'});
137 disp(T)
138 fprintf('Van der Pol Explicit Euler Adaptive Step Size  $\mu=20\n$ ')
139 T = table(reltol, stat2.nfun, stat2.nAccept, stat2.nReject, 'VariableNames',
140 ...

```

```

139    {'Tolerance', 'N. Func', 'N. Accept', 'N. Reject'})};
140 disp(T)
141
142
143 %% Exercise 3.5 Compare using different tolerances and step sizes
144 close all; clear all; clc;
145
146 fs = 20;
147 plot_results = true;
148 mu = [3,20];
149 x0 = [1.0;1.0];
150 h0 = 0.001;
151 tspan = [0 50];
152
153 abstol = [1e-3, 1e-5, 1e-7];
154 reltol = abstol;
155
156 n = length(abstol);
157 % Initialize stat variables
158 acc3 = zeros(n,1); acc20 = zeros(n,1);
159 rej3 = zeros(n,1); rej20 = zeros(n,1);
160 nfun3 = zeros(n,1); nfun20 = zeros(n,1);
161
162 N = 5000;
163
164
165 % Add fixed euler to compare
166 [Tf,Xf] = ImplicitEulerFixedStepSize(@VanderPolfunjac,tspan(1),tspan(2),N,x0
167 ,mu(1));
168 [T2f,X2f] = ImplicitEulerFixedStepSize(@VanderPolfunjac,tspan(1),tspan(2),N,
169 x0,mu(2));
170
171 for i = 1:n
172
173 % Matlab's ode solvers
174 options = odeset("RelTol",reltol(i),"AbsTol",abstol(i));%, 'Stats','on');
175 [Tmu3,Xmu3, statODE45]=ode45(@VanderPolfunjac,tspan,x0,options,mu(1));
176 [Tmu20,Xmu20, statODE45_2]=ode45(@VanderPolfunjac,tspan,x0,options,mu(2)
177 );
178 [T2mu20,X2mu20,statODE15s]=ode15s(@VanderPolfunjac,tspan,x0,options,mu
179 (2));
180 [T2mu3,X2mu3,statODE15s_2]=ode15s(@VanderPolfunjac,tspan,x0,options,mu
181 (1));
182
183 % Using same initial step size as above
184 [T,X,stat] = ImplicitEulerAdaptiveStep(
185     @VanderPolfunjac,tspan,x0,h0,abstol(i),reltol(i),mu(1));
186 [T2,X2,stat2] = ImplicitEulerAdaptiveStep(
187     @VanderPolfunjac,tspan,x0,h0,abstol(i),reltol(i),mu(2));
188
189 if plot_results
190
191     figure, ax=subplot(2,2,1);
192     plot(Tmu3,Xmu3(:,1), 'r', T,X(:,1), 'g', T2mu3,X2mu3(:,1), 'm-',
193          LineWidth',1),

```

```

188 hold on,
189 plot(Tf,Xf(:,1), 'b:', 'LineWidth',1)
190 ylabel('x_1(t)', 'fontsize',fs), xlabel('t', 'fontsize',fs),
191 xlim([0 50])
192
193 ax2=subplot(2,2,2);
194 plot(Tmu3,Xmu3(:,2), 'r',T,X(:,2), 'g',T2mu3,X2mu3(:,2), 'm-',
195 'LineWidth',1),
196 hold on,
197 plot(Tf,Xf(:,2), 'b:', 'LineWidth',1)
198 ylabel('x_2(t)', 'fontsize',fs), xlabel('t', 'fontsize',fs),
199 xlim([0,50])
200 ax3=subplot(2,2,3:4);
201 plot(Xmu3(:,1),Xmu3(:,2), 'r',X(:,1),X(:,2), 'g',X2mu3(:,1),X2mu3(:,2)
202 , 'm-',
203 'LineWidth',1), hold on
204 plot(Xf(:,1),Xf(:,2), 'b:', 'LineWidth',1)
205 xlabel('x_1(t)', 'fontsize',fs), ylabel('x_2(t)', 'fontsize',fs)
206 xlim([-4 4]), ylim([-6 6])
207 legend('ode45', 'ImplicitEuler (Adaptive)', 'ode15s',...
208 'ImplicitEuler (Fixed)', 'Location', 'SouthEast', 'fontsize',13.3)
209 sgttitle({'Van der Pol:  $\mu = 3$ '}, 'fontsize',fs+5),
210 set([ax,ax2,ax3], 'fontsize',fs-3)
211 subtitle({'Tolerance: '+string(abstol(i))}, 'fontsize',15, 'fontweight
212 ', 'bold')
213
214 figure, ax=subplot(2,2,1);
215 plot(Tmu20,Xmu20(:,1), 'r',T2,X2(:,1), 'g',T2mu20,X2mu20(:,1), 'm-',
216 'LineWidth',1),
217 hold on,
218 plot(T2f,X2f(:,1), 'b:', 'LineWidth',1)
219 ylabel('x_1(t)', 'fontsize',fs), xlabel('t', 'fontsize',fs),
220 xlim([0 50])
221
222 ax2=subplot(2,2,2);
223 plot(Tmu20,Xmu20(:,2), 'r',T2,X2(:,2), 'g',T2mu20,X2mu20(:,2), 'm-',
224 'LineWidth',1),
225 hold on,
226 plot(T2f,X2f(:,2), 'b:', 'LineWidth',1)
227 ylabel('x_2(t)', 'fontsize',fs), xlabel('t', 'fontsize',fs),
228 xlim([0,50])
229 ax3=subplot(2,2,3:4);
230 plot(Xmu20(:,1),Xmu20(:,2), 'r',X2(:,1),X2(:,2), 'g',X2mu20(:,1),
231 X2mu20(:,2), 'm-',
232 'LineWidth',1), hold on
233 plot(X2f(:,1),X2f(:,2), 'b:', 'LineWidth',1),
234 xlabel('x_1(t)', 'fontsize',fs), ylabel('x_2(t)', 'fontsize',fs)
235 legend('ode45', 'ImplicitEuler (Adaptive)', 'ode15s',...
236 'ImplicitEuler (Fixed)', 'Location', 'SouthEast', 'fontsize',13.3)
237 sgttitle({'Van der Pol:  $\mu = 20$ '}, 'fontsize',fs+5),
238 subtitle({'Tolerance: '+string(abstol(i))}, 'fontsize',15, 'fontweight
239 ', 'bold')

```

```

236
237     end
238
239 % Store variables
240 statode45(:,i) = statODE45; statode45_2(:,i) = statODE45_2;
241 statode15s(:,i) = statODE15s; statode15s_2(:,i) = statODE15s_2;
242 acc3(i) = stat.nAccept; acc20(i) = stat2.nAccept;
243 rej3(i) = stat.nReject; rej20(i) = stat2.nReject;
244 nfun3(i) = stat.nfun; nfun20(i) = stat2.nfun;
245
246 end
247
248 varnames = {'Tolerance', 'N. Func', 'N. Accept', 'N. Reject'};
249 fprintf('Van der Pol Implicit Euler Adaptive Step Size  $\mu=3$ \n')
250 T=table(reltol', nfun3, acc3, rej3, 'VariableNames', varnames);
251 disp(T)
252 fprintf('Van der Pol Implicit Euler Adaptive Step Size  $\mu=20$ \n')
253 T=table(reltol', nfun20, acc20, rej20, 'VariableNames', varnames);
254 disp(T)
255 fprintf('Van der Pol ODE45  $\mu=3$ \n')
256 T=table(reltol', statode45(3,:)', statode45(1,:)', statode45(2,:)', ...
257     'VariableNames', varnames);
258 disp(T)
259 fprintf('Van der Pol ODE45  $\mu=20$ \n')
260 T=table(reltol', statode45_2(3,:)', statode45_2(1,:)', statode45_2(2,:)', ...
261     'VariableNames', varnames);
262 disp(T)
263 fprintf('Van der Pol ODE15s  $\mu=3$ \n')
264 T=table(reltol', statode15s(3,:)', statode15s_2(1,:)', statode15s_2(2,:)', ...
265     'VariableNames', varnames);
266 disp(T)

```

Listing A.18: Driver for problem 3 on the implicit Euler method.

A.2.4 Problem 4 Driver

```

1 %% Problem 4: Solvers for SDEs
2 %% Exercise 4.2. Explicit-Explicit SDE Solver Van der Pol problem
3 close all; clear all; clc;
4
5 nw = 1;
6 N = [1000,10000]; % Number of time steps
7 Ns = 5; % Number of realizations
8 seed = 100;
9 mu = [3,20];
10 sigma = 0.5; % also try sigma = 1;
11 x0 = [0.5; 0.5];

```

```

12
13 for j=1:2
14
15 %tf = 5*mu(j);
16 tf = 5*mu(j);
17
18 k = 1;
19 p = [mu(j); sigma];
20 for nsteps = N
21 % Get standard wiener process parameter (random variable)
22 [W,T, ]=StdWienerProcess(tf ,nsteps ,nw,Ns,seed );
23
24 % Explicit -Explicit method: STATE INDEPENDENT DIFFUSION
25 X = zeros(length(x0),nsteps+1,Ns);
26 for i=1:Ns
27 X(:,:,i) = SDEsolverExplicitExplicit(...
28 @VanderPolDrift ,@VanderPolDiffusion1 ,...
29 T,x0,W(:,:,i),p);
30 end
31 Xd = SDEsolverExplicitExplicit(...
32 @VanderPolDrift ,@VanderPolDiffusion1 ,...
33 T,x0,W(:,:,i),[mu(j); 0.0]); % without sigma - to show only the
drift term
34
35 % Explicit -Explicit method: STATE DEPENDENT DIFFUSION
36 for i=1:Ns
37 X2(:,:,i) = SDEsolverExplicitExplicit(...
38 @VanderPolDrift ,@VanderPolDiffusion2 ,...
39 T,x0,W(:,:,i),p);
40 end
41 Xd2 = SDEsolverExplicitExplicit(...
42 @VanderPolDrift ,@VanderPolDiffusion2 ,...
43 T,x0,W(:,:,i),[mu(j); 0.0]);
44
45 % Plot results
46 title = {'Explicit -Explicit SDE: Van der Pol','State Independent
Diffusion'};
47 title2 = {'Explicit -Explicit SDE: Van der Pol','State Dependent
Diffusion'};
48
49 fig(k,j) = plotVanDerPolResults(T,X,Xd,title);
50 fig2(k,j) = plotVanDerPolResults(T,X2,Xd2,title2);
51 k = k+1;
52 clear X X2 Xd Xd2
53 end
54 end
55
56
57 %% Exercise 4.3. Implicit -Explicit SDE solver Van der Pol
58 close all; clear all; clc;
59
60 nw = 1;
61 N = [1000,10000]; % Number of time steps
62 Ns = 5; % Number of realizations
63 seed = 100;

```

```

64 mu = [3 ,20];
65 sigma = 0.5;
66 x0 = [0.5; 0.5];
67
68 for j=1:2
69
70 %tf = 5*mu(j);
71 tf = 5*mu(j);
72
73 k = 1;
74 p = [mu(j); sigma];
75 for nsteps = N
76 % Get standard wiener process parameter (random variable)
77 [W,T, ]=StdWienerProcess(tf ,nsteps ,nw,Ns,seed);
78
79
80 % Implicit - Explicit method: STATE INDEPENDENT DIFFUSION
81
82 X = zeros(length(x0) ,nsteps+1,Ns);
83 for i=1:Ns
84 X(: ,: ,i) = SDEsolverImplicitExplicit(...
85 @VanderPolDrift ,@VanderPolDiffusion1 ,...
86 T,x0,W(: ,: ,i),p);
87 end
88 Xd = SDEsolverImplicitExplicit(...
89 @VanderPolDrift ,@VanderPolDiffusion1 ,...
90 T,x0,W(: ,: ,i),[mu(j); 0.0]); % without sigma - to show only the
91 % drift term
92
93
94 % Implicit - Explicit method: STATE DEPENDENT DIFFUSION
95 for i=1:Ns
96 X2(: ,: ,i) = SDEsolverImplicitExplicit(...
97 @VanderPolDrift ,@VanderPolDiffusion2 ,...
98 T,x0,W(: ,: ,i),p);
99 end
100 Xd2 = SDEsolverImplicitExplicit(...
101 @VanderPolDrift ,@VanderPolDiffusion2 ,...
102 T,x0,W(: ,: ,i),[mu(j); 0.0]);
103
104 % Plot results
105 title = {'Implicit - Explicit SDE: Van der Pol','State Independent';
106 Diffusion'};
107 title2 = {'Implicit - Explicit SDE: Van der Pol','State Dependent';
108 Diffusion'};
109
110 fig(k,j) = plotVanDerPolResults(T,X,Xd,title);
111 fig2(k,j) = plotVanDerPolResults(T,X2,Xd2,title2);
112 k = k+1;
113 clear X X2 Xd Xd2
114 end
115 end

```

Listing A.19: Driver for problem 4 on the SDE methods.

A.2.5 Problem 5 Driver

```

1  %% Problem 5: Classical Runge Kutta
2
3  %% Exercise 5.2-5.4 Classical Runge-Kutta method with fixed step size
4  % on the Van der Pol problem
5  close all; clear all;clc
6
7  %Test your algorithms on the van der pol problem (mu=3, mu=20, x0=[1,1]')
8
9  mu = [3,20];
10 tspan = [0,50];
11 x0 = [1,1]';
12
13
14 for N = [3000,10000]
15     h0 = tspan(2)/N;
16
17 [T,X]=ClassicalRungeKuttaSolver(@VanderPolfunjac,tspan,x0,h0,mu(1));
18
19 [T2,X2]=ClassicalRungeKuttaSolver(@VanderPolfunjac,tspan,x0,h0,mu(2));
20
21 % Plot results in two figures for each mu
22 fs = 22; % fontsize for labels
23 figure; f = gcf;
24 ax=subplot(3,1,1);
25 plot(T,X(:,1), 'r', 'LineWidth',1.5);
26 xlabel('t', 'FontSize',fs), ylabel('x_1(t)', 'FontSize',fs),
27 ylim([-3 3]), xlim([0 50]), set(gca, 'fontsize',fs-3)
28 subtitle('Van der Pol:  $\mu=3$ ', 'fontsize',fs-2, 'fontweight', 'bold')
29 %title('Explicit Euler Fixed Step Size')
30 ax2=subplot(3,1,2);
31 plot(T,X(:,2), 'r', 'LineWidth',1.5);
32 xlabel('t', 'FontSize',fs), ylabel('x_2(t)', 'FontSize',fs),
33 ylim([-6 6]), xlim([0 50]), set(gca, 'fontsize',fs-3)
34 ax3=subplot(3,1,3);
35 plot(X(:,1),X(:,2), 'r', 'LineWidth',1.5),
36 ylabel('x_2(t)', 'FontSize',fs), xlabel('x_1(t)', 'FontSize',fs), ylim([-8
8])
37 sgttitle({'Classical Runge-Kutta - Fixed Step'}, 'FontSize', fs+4)
38
39 figure; f2 = gcf;
40 ax4=subplot(3,1,1);
41 plot(T2,X2(:,1), 'r', 'LineWidth',1.5);
42 xlabel('t', 'FontSize',fs), ylabel('x_1(t)', 'FontSize',fs),
43 ylim([-3 3]), xlim([0 50]), set(gca, 'fontsize',fs-3)
44 subtitle('Van der Pol:  $\mu=20$ ', 'fontsize',fs-2, 'fontweight', 'bold')
45 ax5=subplot(3,1,2);
46 plot(T2,X2(:,2), 'r', 'LineWidth',1.5),
47 xlabel('t', 'FontSize',fs), ylabel('x_2(t)', 'FontSize',fs),
48 ylim([-30 30]), xlim([0 50]), set(gca, 'fontsize',fs-3)
49 ax6=subplot(3,1,3);
50 plot(X2(:,1),X2(:,2), 'r', 'LineWidth',1.5), ylim([-33 33])

```

```

51 ylabel('x_2(t)', 'FontSize', fs), xlabel('x_1(t)', 'FontSize', fs)
52 sgttitle({'Classical Runge-Kutta - Fixed Step'}, 'FontSize', fs+4)
53 set(gca, 'fontsize', fs-3)
54
55 end
56
57
58 %% Exercise 5.3-4 Classical RK Adaptive Time Step and Error Estimation
59 close all; clear all; clc;
60
61 x0 = [1.0, 1.0]';
62 mu = [3, 20];
63 abstol = 1e-5;
64 reltol = 1e-5;
65 t0 = 0;
66 fs = 22;
67 tspan = [0, 50];
68
69 h0 = 0.001; % h0 = 10^-2;
70
71 [T,X,stat]=ClassicalRungeKuttaAdaptiveStep(@VanderPolfunjac,tspan,x0,h0,
    abstol,reltol,mu(1));
72 [T2,X2,stat2]=ClassicalRungeKuttaAdaptiveStep(@VanderPolfunjac,tspan,x0,h0,
    abstol,reltol,mu(2));
73
74
75 % Plot results in two figures for each mu
76 figure; f3 = gcf;
77 ax = subplot(3,1,1);
78 plot(T,X(:,1), 'r', 'LineWidth', 1.5);
79 xlabel('t', 'FontSize', fs), ylabel('x_1(t)', 'FontSize', fs),
80 ylim([-3 3]), xlim([0 50])
81 subtitle({'Van Der Pol:  $\mu=3$ '}, 'fontsize', fs-2, 'fontweight', 'bold')
82
83 ax2 = subplot(3,1,2);
84 plot(T,X(:,2), 'r', 'LineWidth', 1.5);
85 xlabel('t', 'FontSize', fs), ylabel('x_2(t)', 'FontSize', fs),
86 ylim([-6 6]), xlim([0 50])
87 ax3 = subplot(3,1,3);
88 plot(X(:,1),X(:,2), 'r', 'LineWidth', 1.5), ylim([-8 8]),
89 ylabel('x_2(t)', 'FontSize', fs), xlabel('x_1(t)', 'FontSize', fs)
90 sgttitle({'Classical Runge-Kutta - Adaptive Step'}, 'FontSize', fs+4)
91
92
93 figure; f4=gcf;
94 ax4=subplot(3,1,1);
95 line3 = plot(T2,X2(:,1), 'r', 'LineWidth', 1.5); ylabel('x_1(t)', 'FontSize', fs)
96 xlabel('t', 'FontSize', fs), ylim([-3 3]), xlim([0 50])
97 subtitle({'Van Der Pol:  $\mu=20$ '}, 'fontsize', fs-2, 'fontweight', 'bold')
98 ax5 = subplot(3,1,2);
99 plot(T2,X2(:,2), 'r', 'LineWidth', 1.5), ylabel('x_2(t)', 'FontSize', fs),
100 xlabel('t', 'FontSize', fs), ylim([-30 30]), xlim([0 50])
101 ax6 = subplot(3,1,3);
102 plot(X2(:,1),X2(:,2), 'r', 'LineWidth', 1.5), ylim([-30 30]),

```

```

103 ylabel('x_2(t)', 'FontSize', fs), xlabel('x_1(t)', 'FontSize', fs)
104 sgtitle({'Classical Runge-Kutta - Adaptive Step'}, 'FontSize', fs+4)
105 set([ax, ax2, ax3, ax4, ax5, ax6], 'fontsize', fs-3)
106
107
108 % Look at error estimation and step sizes
109 figure; f5=gcf;
110 ax=subplot(2,1,1);
111 plot(linspace(0,50, length(stat.r)), stat.r, 'r', 'LineWidth', 1),
112 xlabel('t', 'fontsize', fs), ylabel('r', 'fontsize', fs), ylim([0 1.1])
113 subtitle({'Van Der Pol:  $\mu=3$ '}, 'fontsize', fs-3, 'fontWeight', 'bold')
114 ax2=subplot(2,1,2);
115 plot(linspace(0,50, length(stat.h)), stat.h, 'r', 'LineWidth', 1),
116 xlabel('t'), ylabel('h'), ylim([-0.01 0.04])
117 sgtitle('Classical Runge-Kutta - Adaptive Step', 'fontsize', fs+4)
118
119 figure; f6=gcf;
120 ax3=subplot(2,1,1);
121 plot(linspace(0,50, length(stat2.r)), stat2.r, 'r', 'LineWidth', 1),
122 xlabel('t', 'fontsize', fs), ylabel('r', 'fontsize', fs-2), ylim([0 1.1])
123 subtitle({'Van Der Pol:  $\mu=20$ '}, 'fontsize', fs-3, 'fontWeight', 'bold')
124 ax4=subplot(2,1,2);
125 plot(linspace(0,50, length(stat2.h)), stat2.h, 'r', 'LineWidth', 1),
126 xlabel('t', 'fontsize', fs), ylabel('h', 'fontsize', fs), ylim([0 0.2])%
127 sgtitle('Classical Runge-Kutta - Adaptive Step', 'fontsize', fs+4)
128
129 set([ax, ax2, ax3, ax4], 'fontsize', fs-3)
130
131 fprintf('Van der Pol Classical Runge-Kutta Adaptive Step Size  $\mu=3$ \n')
132 T = table(reltol, stat.nfun, stat.nAccept, stat.nReject, 'VariableNames',
133 ...
134 {'Tolerance', 'N. Func', 'N. Accept', 'N. Reject'});
135 disp(T)
136 fprintf('Van der Pol Classical Runge-Kutta Adaptive Step Size  $\mu=20$ \n')
137 T = table(reltol, stat2.nfun, stat2.nAccept, stat2.nReject, 'VariableNames',
138 ...
139 {'Tolerance', 'N. Func', 'N. Accept', 'N. Reject'});
140 disp(T)
141
142 %% Exercise 5.5 Compare using different tolerances and step sizes
143 %% with ODE solvers from Matlab
144 close all; clear all; clc;
145
146 fs = 22;
147 plot_results = true; % Change to not plot figures
148 mu = [3,20];
149 x0 = [1.0;1.0];
150 h0 = 0.001;
151 tspan = [0 50];
152 abstol = [1e-3, 1e-7, 1e-12];
153 reltol = abstol;
154 n = length(abstol);

```

```

156 % Initialize stat variables
157 acc3 = zeros(n,1); acc20 = zeros(n,1);
158 rej3 = zeros(n,1); rej20 = zeros(n,1);
159 nfun3 = zeros(n,1); nfun20 = zeros(n,1);
160
161 % Add fixed RK4 to compare
162 N = 5000;
163 [Tf,Xf]=ClassicalRungeKuttaSolver(@VanderPolfunjac,tspan,x0,tspan(2)/N,mu(1));
164 [T2f,X2f]=ClassicalRungeKuttaSolver(@VanderPolfunjac,tspan,x0,tspan(2)/N,mu(2));
165
166 for i = 1:n
167
168 % Matlab's ode solvers
169 options = odeset("RelTol",reltol(i),"AbsTol",abstol(i));%, 'Stats ', 'on');
170 [Tmu3,Xmu3, statODE45]=ode45(@VanderPolfunjac,tspan,x0,options,mu(1));
171 [Tmu20,Xmu20, statODE45_2]=ode45(@VanderPolfunjac,tspan,x0,options,mu(2));
172
173 [T2mu20,X2mu20,statODE15s]=ode15s(@VanderPolfunjac,tspan,x0,options,mu(2));
174 [T2mu3,X2mu3,statODE15s_2]=ode15s(@VanderPolfunjac,tspan,x0,options,mu(1));
175
176 % Using same initial step size as above for DOPRI54
177 [T,X,stat]=ClassicalRungeKuttaAdaptiveStep(@VanderPolfunjac,tspan,x0,h0,
178 abstol(i),reltol(i),mu(1));
179 [T2,X2,stat2]=ClassicalRungeKuttaAdaptiveStep(@VanderPolfunjac,tspan,x0,
180 h0,abstol(i),reltol(i),mu(2));
181
182 if plot_results
183
184 figure, ax=subplot(2,2,1);
185 plot(Tmu3,Xmu3(:,1), 'r', T,X(:,1), 'g', T2mu3,X2mu3(:,1), 'm-',
186 'LineWidth',1),
187 hold on,
188 plot(Tf,Xf(:,1), 'b:', 'LineWidth',1)
189 ylabel('x_1(t)', 'fontsize', fs), xlabel('t', 'fontsize', fs),
190 xlim([0 50])
191
192 ax2=subplot(2,2,2);
193 plot(Tmu3,Xmu3(:,2), 'r', T,X(:,2), 'g', T2mu3,X2mu3(:,2), 'm-',
194 'LineWidth',1),
195 hold on,
196 plot(Tf,Xf(:,2), 'b:', 'LineWidth',1)
197 ylabel('x_2(t)', 'fontsize', fs), xlabel('t', 'fontsize', fs),
198 xlim([0,50])
199 ax3=subplot(2,2,3:4);
200 plot(Xmu3(:,1),Xmu3(:,2), 'r', X(:,1),X(:,2), 'g', X2mu3(:,1),X2mu3(:,2)
201 , 'm-',
202 'LineWidth',1), hold on
203 plot(Xf(:,1),Xf(:,2), 'b:', 'LineWidth',1),
204 xlim([-4 4]), ylim([-6 6])
205 xlabel('x_1(t)', 'fontsize', fs), ylabel('x_2(t)', 'fontsize', fs)
206 legend('ode45', 'RK4 (Adaptive)', 'ode15s',...

```

```

201 'RK4 (Fixed)', 'Location', 'SouthEast', 'fontsize', 12)
202 sgtitle({{'Van der Pol:  $\mu = 3$ '}, 'fontsize', fs+5),
203 set([ax, ax2, ax3], 'fontsize', fs-3)
204 subtitle({{'Tolerance: '+string(abstol(i))}, 'fontsize', 15, 'fontweight
205 ', 'bold'})
206
207 figure, ax=subplot(2,2,1);
208 plot(Tmu20, Xmu20(:,1), 'r', T2, X2(:,1), 'g', T2mu20, X2mu20(:,1), 'm--',
209 'LineWidth', 1),
210 hold on,
211 plot(T2f, X2f(:,1), 'b:', 'LineWidth', 1)
212 ylabel('x_1(t)', 'fontsize', fs), xlabel('t', 'fontsize', fs),
213 xlim([0 50])
214
215 ax2=subplot(2,2,2);
216 plot(Tmu20, Xmu20(:,2), 'r', T2, X2(:,2), 'g', T2mu20, X2mu20(:,2), 'm--',
217 'LineWidth', 1),
218 hold on,
219 plot(T2f, X2f(:,2), 'b:', 'LineWidth', 1)
220 ylabel('x_2(t)', 'fontsize', fs), xlabel('t', 'fontsize', fs),
221 xlim([0,50])
222 ax3=subplot(2,2,3:4);
223 plot(Xmu20(:,1), Xmu20(:,2), 'r', X2(:,1), X2(:,2), 'g', X2mu20(:,1),
224 X2mu20(:,2), 'm--', ...
225 'LineWidth', 1), hold on
226 plot(X2f(:,1), X2f(:,2), 'b:', 'LineWidth', 1)
227 xlim([-3 3])
228 xlabel('x_1(t)', 'fontsize', fs), ylabel('x_2(t)', 'fontsize', fs)
229 legend('ode45', 'RK4 (Adaptive)', 'ode15s',
230 'RK4 (Fixed)', 'Location', 'SouthEast', 'fontsize', 12)
231 sgtitle({{'Van der Pol:  $\mu = 20$ '}, 'fontsize', fs+5),
232 set([ax, ax2, ax3], 'fontsize', fs-3)
233 subtitle({{'Tolerance: '+string(abstol(i))}, 'fontsize', 15, 'fontweight
234 ', 'bold'})
235
236 end
237
238 % Store variables
239 statode45(:, i) = statODE45; statode45_2(:, i) = statODE45_2;
240 statode15s(:, i) = statODE15s; statode15s_2(:, i) = statODE15s_2;
241 acc3(i) = stat.nAccept; acc20(i) = stat2.nAccept;
242 rej3(i) = stat.nReject; rej20(i) = stat2.nReject;
243 nfun3(i) = stat.nfun; nfun20(i) = stat2.nfun;
244
245 end
246
247 varnames = {'Tolerance', 'N. Func', 'N. Accept', 'N. Reject'};
248 fprintf('Van der Pol Classical Runge-Kutta Adaptive Step Size  $\mu=3\n$ ')
249 T=table(reltol', nfun3, acc3, rej3, 'VariableNames', varnames);
250 disp(T)
251 fprintf('Van der Pol Classical Runge-Kutta Adaptive Step Size  $\mu=20\n$ ')
252 T=table(reltol', nfun20, acc20, rej20, 'VariableNames', varnames);
253 disp(T)
254 fprintf('Van der Pol ODE45  $\mu=3\n$ ')

```

```

250 T=table(reltol',statode45(3,:)',statode45(1,:)',statode45(2,:)', ...
251     VariableNames', varnames);
252 disp(T)
253 fprintf('Van der Pol ODE45 mu=20\n')
254 T=table(reltol',statode45_2(3,:)', statode45_2(1,:)',statode45_2(2,:)', ...
255     VariableNames', varnames);
256 disp(T)
257 fprintf('Van der Pol ODE15s mu=3\n')
258 T=table(reltol',statode15s(3,:)',statode15s_2(1,:)',statode15s_2(2,:)', ...
259     VariableNames', varnames);
260 disp(T)
261 fprintf('Van der Pol ODE15s mu=20\n')
262 T=table(reltol',statode15s_2(3,:)',statode15s(1,:)',statode15s(2,:)', ...
263     VariableNames', varnames);
264 disp(T)

```

Listing A.20: Driver for problem 5 on the classical Runge-Kutta.

A.2.6 Problem 6 Driver

```

1 %% Problem 6: Dormand- Prince 5(4)
2 %% Test problem for the test equation
3 close all; clear all; clc;
4
5 fs = 27; % fontsize
6 x0 = 1;
7 lambda = -1;
8 tspan = [0,10];
9 h0 = 0.01;
10 abstol = 1e-05;
11 reltol = 1e-05;
12
13 testeq = @(t,x,lambda) x.*lambda; % define test equation function
14
15 [T,X,E]=DOPRI54AdaptiveStep(testeq,tspan,x0,h0, ...
16     abstol,reltol,lambda);
17
18 % Plot Analytical solution vs. DOPRI54
19 fig =gcf;
20 tANA = linspace(0,tspan(2),1001);
21 fANA = (exp(lambda*tANA)*x0)';
22 plot(T,X, 'm-o', 'LineWidth',1.2)
23 hold on
24 plot(tANA, fANA, 'r', 'LineWidth',1),
25 legend('DOPRI54','Analytical solution')
26 title({'The Test Equation'},'FontSize',fs)
27 xlim([0 tspan(2)])
28 xlabel('t','FontSize',fs-3), ylabel('x(t)','FontSize',fs-3)
29 set(gca,'FontSize',16)
30 set(gcf,'paperorientation','landscape')
31

```

```

32
33 %% Exercise 6.4 Error estimation and checking order of method
34
35 hspan = logspace(-4,1,20);
36 nh = length(hspan);
37
38 abstol = 1e-06;
39 reltol = 1e-06;
40
41 % Initialize error estimation
42 Eest = zeros(length(hspan),1);
43 Eest2 = zeros(length(hspan),1);
44 E = zeros(length(hspan),1);
45
46
47 for i=1:nh
48
49 [Tout,Xout,Eout]=DOPRI54AdaptiveStep...
50     (testeq,[0,hspan(i)],x0,hspan(i),abstol,reltol,lambda);
51 % for step doubling
52 [Tout2,Xout2,Eout2]=DOPRI54AdaptiveStep...
53     (testeq,[0,hspan(i)],x0,hspan(i)/2,abstol,reltol,lambda);
54
55 Eest(i) = Eout(end); % adaptive step
56 Eest2(i) = abs(Xout(end)-Xout2(end)); % step doubling
57 E(i) = abs(Xout(end)-x0*exp(hspan(i)*lambda)); % compared to analytical
58 end
59
60 figure, fig2 = gcf;
61 loglog(hspan, Eest, 'bo-', 'LineWidth', 1.2), hold on
62 loglog(hspan, E, 'r+-', 'LineWidth', 1.2), hold on
63 loglog(hspan, Eest2, 'go-', 'LineWidth', 1.2),
64 xlabel('h', 'fontsize',fs-4), ylabel('|e|', 'fontsize',fs-4)
65 xlim([1e-03 1]), ylim([1e-20 1e05]),
66 title('DOPRI54 - Adaptive Step', 'fontsize',fs-2)
67 legend('Estimated Local Error by Embedding', 'Exact Local Error',...
68     'Estimated Local Error by Step Doubling', 'fontsize',15)
69 % save figure
70 set(gca, 'FontSize',15)
71 set(gcf, 'paperorientation', 'landscape')
72 print(fig2,'Dopri54_OrderError','-dpdf','-fillpage')
73
74 % Check order
75 fit1 = polyfit(log(hspan([1:12])),log(Eest([1:12])),1);
76 fit2 = polyfit(log(hspan([8:12])),log(E([8:12])),1);
77 fit3 = polyfit(log(hspan([9:14])),log(Eest2([9:14])),1);
78
79 fprintf('Exact local error gives order %.2f\n',ceil(fit2(1)))
80 fprintf('Embedded error estimation gives order %.2f\n',ceil(fit1(1)))
81 fprintf('Step doubling gives order %.2f\n',ceil(fit3(1)))
82
83 %% Exercise 6.3 cont. Plot the stability region of the method
84 clear all;
85
86 alpha = -5:0.01:5;

```

```

87 beta = -5:0.01:5;
88
89 nreal = length(alpha);
90 nimag = length(beta);
91
92
93 s = 7; % nr of stages in DOPRI54 method
94
95 % Butcher tableau parameters for DOPRI 54
96 A = zeros(s,s);
97     A(2,1) = 1/5;
98     A(3,1) = 3/40;
99     A(4,1) = 44/45;
100    A(5,1) = 19372/6561;
101    A(6,1) = 9017/3168;
102    A(7,1) = 35/384;
103    A(3,2) = 9/40;
104    A(4,2) = -56/15;
105    A(5,2) = -25360/2187;
106    A(6,2) = -355/33;
107    A(4,3) = 32/9;
108    A(5,3) = 64448/6561;
109    A(6,3) = 46732/5247;
110    A(7,3) = 500/1113;
111    A(5,4) = -212/729;
112    A(6,4) = 49/176;
113    A(7,4) = 125/192;
114    A(6,5) = -5103/18656;
115    A(7,5) = -2187/6784;
116    A(7,6) = 11/84;
117 b = [35/384; 0; 500/1113; 125/192; -2187/6784; 11/84; 0];
118 c = [0; 1/5; 3/10; 4/5; 8/9; 1; 1];
119 d = [71/57600; 0; -71/16695; 71/1920; -17253/339200; 22/525; -1/40];
120
121 I = eye(size(A));
122 e = ones(size(A,1),1);
123
124 % Compute the transfer functions
125 for kreal = 1:nreal
126     for kimag = 1:nimag
127         z = alpha(kreal) + i*beta(kimag);
128         tmp = (I-z*A)\e;
129         R = 1 + z*b'*tmp;
130
131         absR(kimag,kreal) = abs(R);
132
133     end
134 end
135
136 % Visualize results
137 figure
138 fs = 25;
139 imagesc(alpha,beta,absR,[0 1]); % Solution estimate
140 grid on
141 colorbar

```

```
142 axis image
143 axis xy
144 xlabel('Re(z)', 'fontsize', fs - 4)
145 ylabel('Im(z)', 'fontsize', fs - 4)
146 title('|R(z)|', 'fontsize', fs),
147 set(gca, 'fontsize', 15)
148 set(gcf, 'PaperOrientation', 'landscape')
149
150 % Check for A stability
151 lis = 1:nreal;
152 ind_ltzero = lis(alpha < 0);
153
154 dopri_Astable = absR(:, ind_ltzero);
155 max_dopri = max(max(dopri_Astable));
156
157 fprintf('The highest value for Re(z)<0 in the stability region for DOPRI54
158     is: %f \n\n', max_dopri)
159
160 % Is the method L stable?
161 %Lets check some low values...
162 z=-10000-i*10000;
163 tmp=(I-z*A)\e;
164 R_dopri=1+z*b'*tmp;
165 absR_dopri=abs(R_dopri);
166
167 fprintf('The stability region for DOPRI54 when z->inf is: %f \n',absR_dopri)
168
169 %% Test problem for the Van der Pol problem
170 close all, clear all, clc
171
172 fs = 24;
173 abstol = 1e-06;
174 reltol = 1e-06;
175
176 mu = [3, 20];
177 x0 = [1.0; 1.0];
178
179 h0 = 0.001;
180 tspan = [0 50];
181
182 % Try for both mu
183 [Tout_VP,Xout_VP,E,stat]=DOPRI54AdaptiveStep(@VanderPolfunjac,tspan,x0,h0
184     ,...
185     abstol,reltol,mu(1));
186
186 [Tout2_VP,Xout2_VP,E2,stat2]=DOPRI54AdaptiveStep(@VanderPolfunjac,tspan,x0,
187     h0, ...
188     abstol,reltol,mu(2));
189
189 % Plot results
190 figure,
191 ax=subplot(3,1,1);
192 plot(Tout_VP,Xout_VP(:,1), 'r', 'LineWidth', 1.5), %title ('\mu = 3'),
193 subtitle({ 'Van Der Pol: \mu=3'}, 'fontsize', fs - 3, 'fontWeight', 'bold')
```

```

194 ylabel('x_1(t)', 'fontsize', fs-3), xlabel('t', 'fontsize', fs-3), xlim(tspan)
195 ax2=subplot(3,1,2);
196 plot(Tout_VP,Xout_VP(:,2), 'r', 'LineWidth', 1.5), %title ('\mu = 3'),
197 ylabel('x_2(t)', 'fontsize', fs-3), xlabel('t', 'fontsize', fs-3), xlim(tspan)
198 ax3=subplot(3,1,3);
199 plot(Xout_VP(:,1), Xout_VP(:,2), 'r', 'LineWidth', 1.5),
200 xlabel('x_1(t)', 'fontsize', fs-3), ylabel('x_2(t)', 'fontsize', fs-3),
201 ylim([-6 6])
202 sgtitle({'DOPRI54 - Adaptive Step'}, 'FontSize', fs)
203 set([ax,ax2,ax3], 'fontsize', fs-4)
204
205
206 figure,
207 ax=subplot(3,1,1);
208 plot(Tout2_VP,Xout2_VP(:,1), 'r', 'LineWidth', 1.5), %title ('\mu = 20'),
209 subtitle({'Van Der Pol: \mu=20'}, 'fontsize', fs-3, 'fontWeight', 'bold')
210 ylabel('x_1(t)', 'fontsize', fs-3), xlabel('t', 'fontsize', fs-3), xlim(tspan)
211 ax2=subplot(3,1,2);
212 plot(Tout2_VP,Xout2_VP(:,2), 'r', 'LineWidth', 1.5), %title ('\mu = 20'),
213 ylabel('x_2(t)', 'fontsize', fs-3), xlabel('t', 'fontsize', fs-3), xlim(tspan)
214 ax3=subplot(3,1,3);
215 plot(Xout2_VP(:,1), Xout2_VP(:,2), 'r', 'LineWidth', 1.5),
216 xlabel('x_1(t)', 'fontsize', fs-3), ylabel('x_2(t)', 'fontsize', fs-3)
217 sgtitle({'DOPRI54 - Adaptive Step'}, 'FontSize', fs)
218
219 set([ax,ax2,ax3], 'fontsize', fs-4)
220
221 % ----- Error estimation and step sizes -----
222 figure; f5=gcf;
223 ax=subplot(2,1,1);
224 plot(linspace(0,50, length(stat.r)), stat.r, 'r', 'LineWidth', 1),
225 xlabel('t', 'fontsize', fs), ylabel('r', 'fontsize', fs), %ylim([-0.1 1])
226 subtitle({'Van Der Pol: \mu=3'}, 'fontsize', fs-3, 'fontWeight', 'bold')
227 ax2=subplot(2,1,2);
228 plot(linspace(0,50, length(stat.h)), stat.h, 'r', 'LineWidth', 1),
229 xlabel('t'), ylabel('h'),
230 sgtitle('DOPRI54 - Adaptive Step', 'fontsize', fs+4)
231
232 figure; f6=gcf;
233 ax3=subplot(2,1,1);
234 plot(linspace(0,50, length(stat2.r)), stat2.r, 'r', 'LineWidth', 1),
235 xlabel('t', 'fontsize', fs), ylabel('r', 'fontsize', fs-2), %ylim([0 1])
236 subtitle({'Van Der Pol: \mu=20'}, 'fontsize', fs-3, 'fontWeight', 'bold')
237 ax4=subplot(2,1,2);
238 plot(linspace(0,50, length(stat2.h)), stat2.h, 'r', 'LineWidth', 1),
239 xlabel('t', 'fontsize', fs), ylabel('h', 'fontsize', fs), %ylim([-0.01 0.3])%
240 sgtitle('DOPRI54 - Adaptive Step', 'fontsize', fs+4)
241
242 set([ax,ax2,ax3,ax4], 'fontsize', fs-3)
243
244 fprintf('Van der Pol Explicit Euler Adaptive Step Size \mu=3\n')
245 T = table(reltol, stat.nfun, stat.nAccept, stat.nReject, 'VariableNames',
246 ...
247 {'Tolerance', 'N. Func', 'N. Accept', 'N. Reject'});
disp(T)

```

```

248 fprintf('Van der Pol Explicit Euler Adaptive Step Size  $\mu=20$ \n')
249 T = table(reltol, stat2.nfun, stat2.nAccept, stat2.nReject, 'VariableNames',
250     ...
251     {'Tolerance','N. Func','N. Accept','N. Reject'});
252 disp(T)
253
254 %% Test your algorithms on the adiabatic CSTR problem described in the
255 %% papers
256 % on learn (3D-version and 1-D version)
257 fs = 24;
258 t = [0 35];
259 T0 = 273.65; % in Kelvin for 1D CSTR
260 x0 = [0,0,T0]'; % for 3D
261 h0 =0.1;
262
263 % Discrete function for flow rate for different times
264
265 reltol = 1e-5;
266 abstol = reltol;
267
268 [T,X,E,stat] = DOPRI54AdaptiveStep(@CSTR_1D,t ,T0,h0 ,reltol ,abstol ,@FlowRate)
269 ;
270 [T3D,X3D,E3D,stat3D] = DOPRI54AdaptiveStep(@CSTR_3D,t ,x0,h0 ,reltol ,abstol ,
271 @FlowRate);
272
273 % Get the flow rate trajectory for the plot
274 count = 0;
275 for t = 0:0.01:35
276     count = count +1;
277     F(count) = FlowRate(t);
278 end
279
280 figure
281 sp=subplot(2,1,1);
282 ax=plot(T,X-273.15, 'g', 'LineWidth',1.2); ylabel('T [°C]', 'fontsize',fs -4);
283 hold on
284 ax2=plot(T3D, X3D(:,3)-273.15,'r-.', 'LineWidth',1.2);
285 legend([ax,ax2], '1D Version', '3D Version', 'Location', 'NorthWest', 'fontsize'
286 ,13)
287 sp2=subplot(2,1,2);
288 plot(0:0.01:35, F, 'r', 'LineWidth',1.2),
289 xlabel('t [min]', 'fontsize',fs -4), ylabel('F [mL/min]', 'fontsize',fs -4)
290 ylim([0 1000])
291 sgttitle('DOPRI54 - Adiabatic CSTR Simulation', 'fontsize',fs -1)
292 set([sp,sp2], 'fontsize',15)
293 set(gcf, 'PaperOrientation', 'landscape')
294
295
296 %% Compare results with ode45 which implements DOPRI54 method and ode15s
297 %% Comparing with the Van der Pol problem stiff vs non-stiff etc.
298 close all; clear all; clc;
299
300 fs = 20;
301 plot_results = true;

```

```

298 mu = [3,20];
299 x0 = [1.0;1.0];
300 h0 = 0.001;
301 tspan = [0 50];
302
303 abstol = [1e-3, 1e-7, 1e-12];
304 reltol = abstol;
305
306 n = length(abstol);
307 % Initialize stat variables
308 acc3 = zeros(n,1); acc20 = zeros(n,1);
309 rej3 = zeros(n,1); rej20 = zeros(n,1);
310 nfun3 = zeros(n,1); nfun20 = zeros(n,1);
311
312 for i = 1:n
313
314 % Matlab's ode solvers
315 options = odeset("RelTol",reltol(i),"AbsTol",abstol(i));%, 'Stats ', 'on');
316 [Tmu3,Xmu3, statODE45]=ode45(@VanderPolfunjac,tspan,x0,options,mu(1));
317 [Tmu20,Xmu20, statODE45_2]=ode45(@VanderPolfunjac,tspan,x0,options,mu(2))
318 );
319 [T2mu20,X2mu20,statODE15s_2]=ode15s(@VanderPolfunjac,tspan,x0,options,mu
320 (2));
321 [T2mu3,X2mu3,statODE15s]=ode15s(@VanderPolfunjac,tspan,x0,options,mu(1))
322 ;
323
324 % Using same initial step size as above for DOPRI54
325 [T,X,E,stat] = DOPRI54AdaptiveStep(@VanderPolfunjac,tspan,x0,h0,abstol(i)
326 ),reltol(i),mu(1));
327 [T2,X2,E2,stat2] = DOPRI54AdaptiveStep(@VanderPolfunjac,tspan,x0,h0,
328 abstol(i),reltol(i),mu(2));
329
330 if plot_results
331
332 figure,
333 ax=subplot(2,2,1);
334 plot(Tmu3,Xmu3(:,1), 'r',T,X(:,1), 'g',T2mu3,X2mu3(:,1), 'm-',
335 'LineWidth',1),
336 ylabel('x_1(t)', 'fontsize',fs), xlabel('t', 'fontsize',fs),
337 xlim([0 50])
338
339 ax2=subplot(2,2,2);
340 plot(Tmu3,Xmu3(:,2), 'r',T,X(:,2), 'g',T2mu3,X2mu3(:,2), 'm-',
341 'LineWidth',1),
342 ylabel('x_2(t)', 'fontsize',fs), xlabel('t', 'fontsize',fs),
343 xlim([0,50])
344
345 ax3=subplot(2,2,3:4);
346 plot(Xmu3(:,1),Xmu3(:,2), 'r',X(:,1),X(:,2), 'g',X2mu3(:,1),X2mu3(:,2)
347 , 'm-',
348 'LineWidth',1),
349 xlabel('x_1(t)', 'fontsize',fs), ylabel('x_2(t)', 'fontsize',fs)
350 xlim([-3 3]), ylim([-6 6])
351 legend('ode45', 'DOPRI54 (Adaptive)', 'ode15s',
352 'Location', 'SouthEast', 'fontsize',12)

```

```

345      sgtitle({ 'Van der Pol:  $\mu = 3$ '}, 'fontsize', fs+5),
346      set([ax ax2 ax3], 'fontsize', fs-3)
347      subtitle({ 'Tolerance: '+string(abstol(i))}, 'fontsize', 15, 'fontweight
348          ', 'bold')
349
350
351      figure, ax=subplot(2,2,1);
352      plot(Tmu20,Xmu20(:,1), 'r', T2,X2(:,1), 'g', T2mu20,X2mu20(:,1), 'm--',
353          'LineWidth', 1),
354      ylabel('x_1(t)', 'fontsize', fs), xlabel('t', 'fontsize', fs),
355      xlim([0 50])
356
357      ax2=subplot(2,2,2);
358      plot(Tmu20,Xmu20(:,2), 'r', T2,X2(:,2), 'g', T2mu20,X2mu20(:,2), 'm--',
359          'LineWidth', 1),
360      ylabel('x_2(t)', 'fontsize', fs), xlabel('t', 'fontsize', fs),
361      xlim([0,50])
362
363      ax3=subplot(2,2,3:4);
364      plot(Xmu20(:,1),Xmu20(:,2), 'r', X2(:,1),X2(:,2), 'g', X2mu20(:,1),
365          X2mu20(:,2), 'm--', ...
366          'LineWidth', 1),
367      xlim([-3 3])
368      xlabel('x_1(t)', 'fontsize', fs), ylabel('x_2(t)', 'fontsize', fs)
369      legend('ode45', 'DOPRI54 (Adaptive)', 'ode15s', ...
370          'Location', 'SouthEast', 'fontsize', 12)
371      sgtitle({ 'Van der Pol:  $\mu = 20$ '}, 'fontsize', fs+5),
372      set([ax,ax2,ax3], 'fontsize', fs-3)
373      subtitle({ 'Tolerance: '+string(abstol(i))}, 'fontsize', 15, 'fontweight
374          ', 'bold')
375
376
377      end
378      % Store variables
379      statode45(:,i) = statODE45; statode45_2(:,i) = statODE45_2;
380      statode15s(:,i) = statODE15s; statode15s_2(:,i) = statODE15s_2;
381      acc3(i) = stat.nAccept; acc20(i) = stat2.nAccept;
382      rej3(i) = stat.nReject; rej20(i) = stat2.nReject;
383      nfun3(i) = stat.nfun; nfun20(i) = stat2.nfun;
384
385
386      end
387
388      varnames = {'Tolerance', 'N. Func', 'N. Accept', 'N. Reject'};
389      fprintf('Van der Pol DOPRI54 Adaptive Step Size  $\mu=3\n$ ');
390      T=table(reltol', nfun3, acc3, rej3, 'VariableNames', varnames);
391      disp(T)
392      fprintf('Van der Pol DOPRI54 Adaptive Step Size  $\mu=20\n$ ')
393      T=table(reltol', nfun20, acc20, rej20, 'VariableNames', varnames);
394      disp(T)
395      fprintf('Van der Pol ODE45  $\mu=3\n$ ')
396      T=table(reltol', statode45(3,:)', statode45(1,:)', statode45(2,:)', ...
397          'VariableNames', varnames);
398      disp(T)
399      fprintf('Van der Pol ODE45  $\mu=20\n$ ')
400      T=table(reltol', statode45_2(3,:)', statode45_2(1,:)', statode45_2(2,:)', ...
401          'VariableNames', varnames);
402      disp(T)

```

```

393 fprintf('Van der Pol ODE15s mu=3\n')
394 T=table(reltol',statode15s(3,:)',statode15s(1,:)',statode15s(2,:)', '
395     VariableNames', varnames);
396 disp(T)
397 fprintf('Van der Pol ODE15s mu=20\n')
398 T=table(reltol',statode15s_2(3,:)',statode15s_2(1,:)',statode15s_2(2,:)', '
399     VariableNames', varnames);
400 disp(T)

```

Listing A.21: Driver for problem 6 on the DOPRI54 method.

A.2.7 Problem 7 Driver

```

1 %% Problem 7: ESDIRK23
2 %% Exercise 7.2 Plot stability region of the ESDIRK23 method
3 close all; clear all; clc;
4 mu=[3,20];
5 x0=[1,1]';
6 tspan=[0 30];
7 h0=0.01;
8
9 absTol = 10^-6;
10 relTol = absTol;
11
12 alpha = -20:0.01:20;
13 beta = -20:0.01:20;
14
15 nreal = length(alpha);
16 nimag = length(beta);
17
18
19 gamma = (2-sqrt(2))/2;
20
21 a21 = gamma;
22 b2 = (1-gamma)/2;
23 b1 = (1-gamma)/2;
24
25 for kreal = 1:nreal
26     for kimag = 1:nimag
27         z = alpha(kreal) + i*beta(kimag);
28         R = (1+(b1+b2-gamma)*z + (a21*b2-b1*gamma)*z^2)/((1-gamma*z)^2);
29         absR(kimag,kreal) = abs(R);
30     end
31 end
32
33 figure
34 fs = 20;
35 imagesc(alpha ,beta ,absR,[0 1]);
36 grid on
37 colorbar
38 axis image

```

```

39 axis xy
40 xlabel('Re(z)', 'fontsize', fs -4)
41 ylabel('Im(z)', 'fontsize', fs -4)
42 title('|R(z)|', 'fontsize', fs),
43 set(gca, 'fontsize', 15)
44 set(gcf, 'PaperOrientation', 'landscape')
45
46 % Check for A stability
47 lis = 1:nreal;
48 ind_ltzero=lis(alpha<0);
49
50 esdirk_Astable = absR(:,ind_ltzero);
51 max_esdirk = max(max(esdirk_Astable));
52
53 fprintf('The highest value for Re(z)<0 in the stability region for DOPRI54
      is: %f \n\n',max_esdirk)
54
55 % Is the method L stable?
56 %Lets check some low values...
57 z=-10000-i*10000;
58 tmp=(1+b1+b2-gamma);
59 tmp2=(a21*b2-b1*gamma)*z^2;
60 R_esdirk=(tmp+tmp2)/((1-gamma*z)^2);
61 absR_esdirk=abs(R_esdirk);
62
63 fprintf('The stability region for ESDIRK23 when z->inf is: %f \n',
      absR_esdirk)
64
65 %% Exercise 7.4 Test algorithm on the Van der Pol problem
66 close all;clear all;clc
67
68 fs = 25;
69 mu=[3,20];
70 x0=[1,1]';
71 tspan=[0 50];
72 h0=0.001;
73
74 absTol = 10^-5;
75 relTol = absTol;
76
77 [Tout_VP,Xout_VP,Gout,info ,stat] = ESDIRK23AdaptiveStep(@VanderPolFun,
      @VanderPolJac,tspan,x0,h0,absTol,relTol,mu(1));
78 [Tout2_VP,Xout2_VP,Gout2,info2 ,stat2] = ESDIRK23AdaptiveStep(@VanderPolFun,
      @VanderPolJac,tspan,x0,h0,absTol,relTol,mu(2));
79
80
81 % Plot results
82 figure ,
83 ax=subplot(3,1,1);
84 plot(Tout_VP,Xout_VP(:,1), 'r', 'LineWidth',1.5), %title ('\mu = 3'),
85 subtitle({{'Van Der Pol: \mu=3'}}, 'fontsize', fs -3, 'fontWeight', 'bold')
86 ylabel('x_1(t)', 'fontsize', fs -3), xlabel('t', 'fontsize', fs -3), xlim(tspan)
87 ax2=subplot(3,1,2);
88 plot(Tout_VP,Xout_VP(:,2), 'r', 'LineWidth',1.5), %title ('\mu = 3'),
89 ylabel('x_2(t)', 'fontsize', fs -3), xlabel('t', 'fontsize', fs -3), xlim(tspan)

```

```

90 ax3=subplot(3,1,3);
91 plot(Xout_VP(:,1), Xout_VP(:,2), 'r', 'LineWidth', 1.5),
92 xlabel('x_1(t)', 'fontsize', fs-3), ylabel('x_2(t)', 'fontsize', fs-3),
93 ylim([-6 6])
94 sgttitle({'ESDIRK23 - Adaptive Step'}, 'FontSize', fs)
95 set([ax,ax2,ax3], 'fontsize', fs-4)
96
97 % Save figure
98 %print('VanderPolESDIRK23Adaptive_mu3', '-dpdf', '-fillpage')
99
100
101 figure,
102 ax=subplot(3,1,1);
103 plot(Tout2_VP,Xout2_VP(:,1), 'r', 'LineWidth', 1.5), %title('\mu = 20'),
104 subtitle({'Van Der Pol: \mu=20'}, 'fontsize', fs-3, 'fontweight', 'bold')
105 ylabel('x_1(t)', 'fontsize', fs-3), xlabel('t', 'fontsize', fs-3), xlim(tspan)
106 ax2=subplot(3,1,2);
107 plot(Tout2_VP,Xout2_VP(:,2), 'r', 'LineWidth', 1.5), %title('\mu = 20'),
108 ylabel('x_2(t)', 'fontsize', fs-3), xlabel('t', 'fontsize', fs-3), xlim(tspan)
109 %sgttitle({'Van der Pol Problem', 'ESDIRK23 with Adaptive Time Step'})
110 ax3=subplot(3,1,3);
111 plot(Xout2_VP(:,1), Xout2_VP(:,2), 'r', 'LineWidth', 1.5),
112 xlim([-3 3])
113 xlabel('x_1(t)', 'fontsize', fs-3), ylabel('x_2(t)', 'fontsize', fs-3)
114 sgttitle({'ESDIRK23 - Adaptive Step'}, 'FontSize', fs)
115 set([ax,ax2,ax3], 'fontsize', fs-4)
116
117
118 % Error estimation and step sizes
119
120
121 % Look at error estimation and step sizes
122 figure; f5=gcf;
123 ax=subplot(2,1,1);
124 plot(linspace(0,50, length(stat.r)), stat.r, 'r', 'LineWidth', 1),
125 xlabel('t', 'fontsize', fs), ylabel('r', 'fontsize', fs), %ylim([-0.1 1])
126 subtitle({'Van Der Pol: \mu=3'}, 'fontsize', fs-3, 'fontweight', 'bold')
127 ax2=subplot(2,1,2);
128 plot(linspace(0,50, length(stat.h)), stat.h, 'r', 'LineWidth', 1),
129 xlabel('t'), ylabel('h'),
130 sgttitle('ESDIRK23 - Adaptive Step', 'fontsize', fs)
131
132 figure; f6=gcf;
133 ax3=subplot(2,1,1);
134 plot(linspace(0,50, length(stat2.r)), stat2.r, 'r', 'LineWidth', 1),
135 xlabel('t', 'fontsize', fs), ylabel('r', 'fontsize', fs-2), %ylim([0 1])
136 subtitle({'Van Der Pol: \mu=20'}, 'fontsize', fs-3, 'fontweight', 'bold')
137 ax4=subplot(2,1,2);
138 plot(linspace(0,50, length(stat2.h)), stat2.h, 'r', 'LineWidth', 1),
139 xlabel('t', 'fontsize', fs), ylabel('h', 'fontsize', fs), %ylim([-0.01 0.3])%
140 sgttitle('ESDIRK23 - Adaptive Step', 'fontsize', fs)
141
142 set([ax,ax2,ax3,ax4], 'fontsize', fs-6)
143
144

```

```

145 fprintf('Van der Pol ESDIRK23 Adaptive Step Size  $\mu=3$ \n')
146 T = table(relTol, info.nFun, info.nAccept, info.nFail, 'VariableNames', ...
147     {'Tolerance', 'N. Func', 'N. Accept', 'N. Reject'});
148 disp(T)
149 fprintf('Van der Pol ESDIRK23 Adaptive Step Size  $\mu=20$ \n')
150 T = table(relTol, info2.nFun, info2.nAccept, info2.nFail, 'VariableNames', ...
151     ...
152     {'Tolerance', 'N. Func', 'N. Accept', 'N. Reject'});
153 disp(T)
154
155 % Exercise 7.5: Compare with Matlabs ODE solvers and other solvers
156 % developed in
157 % this exam problem
158 close all; clear all; clc;
159
160 fs = 20;
161 plot_results = true;
162 mu = [3,20];
163 x0 = [1.0;1.0];
164 h0 = 0.001;
165 tspan = [0 50];
166
167 abstol = [1e-3, 1e-7];
168 reltol = abstol;
169
170 n = length(abstol);
171 % Initialize stat variables
172 acc3 = zeros(n,1); acc20 = zeros(n,1);
173 rej3 = zeros(n,1); rej20 = zeros(n,1);
174 nfun3 = zeros(n,1); nfun20 = zeros(n,1);
175
176 for i = 1:n
177
178     % Matlab's ode solvers
179     options = odeset("RelTol", reltol(i), "AbsTol", abstol(i));%, 'Stats ', 'on' );
180     [Tmu3,Xmu3, statODE45]=ode45(@VanderPolfunjac ,tspan ,x0 ,options ,mu(1));
181     [Tmu20,Xmu20, statODE45_2]=ode45(@VanderPolfunjac ,tspan ,x0 ,options ,mu(2) );
182     [T2mu20,X2mu20,statODE15s_2]=ode15s(@VanderPolfunjac ,tspan ,x0 ,options ,mu(2));
183     [T2mu3,X2mu3,statODE15s]=ode15s(@VanderPolfunjac ,tspan ,x0 ,options ,mu(1));
184
185     % Using same initial step size as above for ESDIRK23
186     [T,X,Gout,info,stat] = ESDIRK23AdaptiveStep(@VanderPolFun ,@VanderPolJac ,
187         tspan ,x0 ,h0 ,abstol(i) ,reltol(i) ,mu(1));
188     [T2,X2,Gout2,info2,stat2] = ESDIRK23AdaptiveStep(@VanderPolFun ,
189         @VanderPolJac,tspan ,x0 ,h0 ,abstol(i) ,reltol(i) ,mu(2));
190
191 if plot_results
192
193     figure ,
194     ax=subplot(2,2,1);

```

```

192 plot(Tmu3,Xmu3(:,1), 'r',T,X(:,1), 'g',T2mu3,X2mu3(:,1), 'm--', ...
193     'LineWidth',1),
194 ylabel('x_1(t)', 'fontsize',fs), xlabel('t', 'fontsize',fs),
195 xlim([0 50])
196 ax2=subplot(2,2,2);
197 plot(Tmu3,Xmu3(:,2), 'r',T,X(:,2), 'g',T2mu3,X2mu3(:,2), 'm--', ...
198     'LineWidth',1),
199 ylabel('x_2(t)', 'fontsize',fs), xlabel('t', 'fontsize',fs),
200 xlim([0,50])
201 ax3=subplot(2,2,3:4);
202 plot(Xmu3(:,1),Xmu3(:,2), 'r',X(:,1),X(:,2), 'g',X2mu3(:,1),X2mu3(:,2) ...
203     , 'm--', ...
204     'LineWidth',1),
205 xlabel('x_1(t)', 'fontsize',fs), ylabel('x_2(t)', 'fontsize',fs)
206 xlim([-3 3]), ylim([-6 6])
207 legend('ode45', 'ESDIRK23 (Adaptive)', 'ode15s',...
208     'Location', 'SouthEast', 'fontsize',12)
209 sgttitle({ 'Van der Pol:  $\mu = 3$ '}, 'fontsize',fs+5),
210 set([ax ax2 ax3], 'fontsize',fs-3)
211 subtitle({ 'Tolerance: '+string(abstol(i))}, 'fontsize',15, 'fontweight' ...
212     , 'bold')
213
214 figure, ax=subplot(2,2,1);
215 plot(Tmu20,Xmu20(:,1), 'r',T2,X2(:,1), 'g',T2mu20,X2mu20(:,1), 'm--', ...
216     'LineWidth',1),
217 ylabel('x_1(t)', 'fontsize',fs), xlabel('t', 'fontsize',fs),
218 xlim([0 50])
219 ax2=subplot(2,2,2);
220 plot(Tmu20,Xmu20(:,2), 'r',T2,X2(:,2), 'g',T2mu20,X2mu20(:,2), 'm--', ...
221     'LineWidth',1),
222 ylabel('x_2(t)', 'fontsize',fs), xlabel('t', 'fontsize',fs),
223 xlim([0,50])
224 ax3=subplot(2,2,3:4);
225 plot(Xmu20(:,1),Xmu20(:,2), 'r',X2(:,1),X2(:,2), 'g',X2mu20(:,1), ...
226     X2mu20(:,2), 'm--', ...
227     'LineWidth',1),
228 xlabel('x_1(t)', 'fontsize',fs), ylabel('x_2(t)', 'fontsize',fs)
229 legend('ode45', 'ESDIRK23 (Adaptive)', 'ode15s',...
230     'Location', 'SouthEast', 'fontsize',12)
231 sgttitle({ 'Van der Pol:  $\mu = 20$ '}, 'fontsize',fs+5),
232 set([ax,ax2,ax3], 'fontsize',fs-3)
233 subtitle({ 'Tolerance: '+string(abstol(i))}, 'fontsize',15, 'fontweight' ...
234     , 'bold')
235
236 end
237 % Store variables
238 statode45(:,i) = statODE45; statode45_2(:,i) = statODE45_2;
239 statode15s(:,i) = statODE15s; statode15s_2(:,i) = statODE15s_2;
240 acc3(i) = info.nAccept; acc20(i) = info2.nAccept;
241 rej3(i) = info.nFail; rej20(i) = info2.nFail;
242 nfun3(i) = info.nFun; nfun20(i) = info2.nFun;
243
244 end

```

```

239 varnames = {'Tolerance', 'N. Func', 'N. Accept', 'N. Reject'};
240 fprintf('Van der Pol ESDIRK23 Adaptive Step Size  $\mu=3$ \n');
241 T=table(reltol', nfun3, acc3, rej3, 'VariableNames', varnames);
242 disp(T)
243 fprintf('Van der Pol ESDIRK23 Adaptive Step Size  $\mu=20$ \n')
244 T=table(reltol', nfun20, acc20, rej20, 'VariableNames', varnames);
245 disp(T)
246 fprintf('Van der Pol ODE45  $\mu=3$ \n')
247 T=table(reltol', statode45(3,:)', statode45(1,:)', statode45(2,:)', ...
248     'VariableNames', varnames);
249 disp(T)
250 fprintf('Van der Pol ODE45  $\mu=20$ \n')
251 T=table(reltol', statode45_2(3,:)', statode45_2(1,:)', statode45_2(2,:)', ...
252     'VariableNames', varnames);
253 disp(T)
254 fprintf('Van der Pol ODE15s  $\mu=3$ \n')
255 T=table(reltol', statode15s(3,:)', statode15s(1,:)', statode15s(2,:)', ...
256     'VariableNames', varnames);
257 disp(T)
258
259 %% Exercise 7.5 cont. Comparing with the other solvers developed in this
260 %% exam problems
261 close all; clear all; clc;
262
263 fs = 20;
264 plot_results = true;
265 mu = [3,20];
266 x0 = [1.0;1.0];
267 h0 = 0.001;
268 tspan = [0 50];
269
270 abstol = [1e-3, 1e-7];
271 reltol = abstol;
272 n = length(abstol);
273
274 % Initialize stat variables
275 acc3 = zeros(n,1); acc20 = zeros(n,1);
276 rej3 = zeros(n,1); rej20 = zeros(n,1);
277 nfun3 = zeros(n,1); nfun20 = zeros(n,1);
278
279 for i = 1:n
280
281     % Classical Runge-Kutta
282     [Tmu3,Xmu3, statRK4] = ClassicalRungeKuttaAdaptiveStep(@VanderPolfunjac,
283                 tspan,x0,h0,abstol(i),reltol(i),mu(1));
284     [Tmu20,Xmu20, statRK42] = ClassicalRungeKuttaAdaptiveStep(
285                 @VanderPolfunjac,tspan,x0,h0,abstol(i),reltol(i),mu(2));
286
287     % Explicit Euler
288     [T2mu3,X2mu3,statEE] = ExplicitEulerAdaptiveTimeStep(...

```

```

287     @VanderPolfunjac ,tspan ,x0,h0, abstol(i) ,reltol(i) ,mu(1));
288 [T2mu20,X2mu20,statEE2] = ExplicitEulerAdaptiveTimeStep(...
289     @VanderPolfunjac ,tspan ,x0,h0, abstol(i) ,reltol(i) ,mu(2));
290
291 % Implicit Euler
292 [TIEmu3,XIEmu3, statIE]= ImplicitEulerAdaptiveStep(@VanderPolfunjac ,
293     tspan ,x0,h0, abstol(i) ,reltol(i) ,mu(1));
294 [TIEmu20,XIEmu20, statIE2]= ImplicitEulerAdaptiveStep(@VanderPolfunjac ,
295     tspan ,x0,h0, abstol(i) ,reltol(i) ,mu(2));
296
297 % Doprri54
298 [TD, XD, ED, statD] = DOPRI54AdaptiveStep(@VanderPolfunjac ,tspan ,x0,h0,
299     abstol(i) ,reltol(i) ,mu(1));
300 [TD2, XD2, ED2, statD2] = DOPRI54AdaptiveStep(@VanderPolfunjac ,tspan ,x0,
301     h0, abstol(i) ,reltol(i) ,mu(2));
302
303 % ESDIRK23
304 [T,X,Gout ,info ,stat] = ESDIRK23AdaptiveStep(@VanderPolFun ,@VanderPolJac ,
305     tspan ,x0,h0, abstol(i) ,reltol(i) ,mu(1));
306 [T2,X2,Gout ,info2 ,stat2] = ESDIRK23AdaptiveStep(@VanderPolFun ,
307     @VanderPolJac ,tspan ,x0,h0, abstol(i) ,reltol(i) ,mu(2));
308
309 if plot_results
310
311     figure ,
312     ax=subplot(2,2,1);
313     plot(Tmu3,Xmu3(:,1) , 'r' ,T,X(:,1) , 'g' ,T2mu3,X2mu3(:,1) , 'm-' ,
314         'LineWidth' ,1),
315     hold on, plot(TIEmu3,XIEmu3(:,1) , 'b:' ,TD, XD(:,1) , 'k' , 'LineWidth' ,1)
316     ylabel('x_1(t)' , 'fontsize' ,fs) , xlabel('t' , 'fontsize' ,fs) ,
317     xlim([0 50])
318
319     ax2=subplot(2,2,2);
320     plot(Tmu3,Xmu3(:,2) , 'r' ,T,X(:,2) , 'g' ,T2mu3,X2mu3(:,2) , 'm-' ,
321         'LineWidth' ,1),
322     hold on, plot(TIEmu3,XIEmu3(:,2) , 'b:' ,TD, XD(:,2) , 'k' , 'LineWidth' ,1)
323     ylabel('x_2(t)' , 'fontsize' ,fs) , xlabel('t' , 'fontsize' ,fs) ,
324     xlim([0 ,50])
325
326     ax3=subplot(2,2,3:4);
327     plot(Xmu3(:,1) ,Xmu3(:,2) , 'r' ,X(:,1) ,X(:,2) , 'g' ,X2mu3(:,1) ,X2mu3(:,2)
328         , 'm-' ,...
329         'LineWidth' ,1), hold on,
330     plot(XIEmu3(:,1) ,XIEmu3(:,2) , 'b:' ,XD(:,1) , XD(:,2) , 'k' , 'LineWidth'
331         ,1)
332     xlabel('x_1(t)' , 'fontsize' ,fs) , ylabel('x_2(t)' , 'fontsize' ,fs)
333     xlim([-3 3]), ylim([-6 6])
334     legend('RK4' , 'ESDIRK23' , 'ExplicitEuler' , 'ImplicitEuler' , 'DOPRI54'
335         ,...
336         'Location' , 'northwest' , 'fontsize' ,12.7)
337     sgtitle({ 'Van der Pol:  $\mu = 3$ '}, 'fontsize' ,fs+5),
338     set([ax ax2 ax3] , 'fontsize' ,fs -3)

```

```

330    subtitle({ 'Tolerance: '+string(abstol(i))}, 'fontsize',15, 'fontweight
331      ', 'bold')
332
333    figure , ax=subplot(2,2,1);
334    plot(Tmu20,Xmu20(:,1), 'r' ,T2,X2(:,1) , 'g' ,T2mu20,X2mu20(:,1) , 'm-' ,
335      'LineWidth',1),
336    hold on, plot(TIEmu20,XIEmu20(:,1) , 'b:' ,TD2, XD2(:,1) , 'k' , 'LineWidth
337      ',1)
338    ylabel('x_1(t)' , 'fontsize' ,fs) , xlabel('t' , 'fontsize' ,fs) ,
339    xlim([0 50])
340
341    ax2=subplot(2,2,2);
342    plot(Tmu20,Xmu20(:,2) , 'r' ,T2,X2(:,2) , 'g' ,T2mu20,X2mu20(:,2) , 'm-' ,
343      'LineWidth',1),
344    hold on, plot(TIEmu20,XIEmu20(:,2) , 'b:' ,TD2, XD2(:,2) , 'k' , 'LineWidth
345      ',1)
346    ylabel('x_2(t)' , 'fontsize' ,fs) , xlabel('t' , 'fontsize' ,fs) ,
347    xlim([0 ,50])
348    ax3=subplot(2,2,3:4);
349    plot(Xmu20(:,1) ,Xmu20(:,2) , 'r' ,X2(:,1) ,X2(:,2) , 'g' ,X2mu20(:,1) ,
350      X2mu20(:,2) , 'm-' ,...
351      'LineWidth',1), hold on,
352    plot(XIEmu20(:,1) ,XIEmu20(:,2) , 'b:' ,XD2(:,1) , XD2(:,2) , 'k' ,
353      'LineWidth',1)
354    xlim([-3 3])
355    xlabel('x_1(t)' , 'fontsize' ,fs) , ylabel('x_2(t)' , 'fontsize' ,fs)
356    legend('RK4' , 'ESDIRK23' , 'ExplicitEuler' , 'ImplicitEuler' , 'DOPRI54'
357          , ...
358          'Location' , 'northwest' , 'fontsize' ,12.7)
359    sgttitle({ 'Van der Pol:  $\mu = 20$ '}, 'fontsize' ,fs+5),
360    set([ax,ax2,ax3] , 'fontsize' ,fs-3)
361    subtitle({ 'Tolerance: '+string(abstol(i))}, 'fontsize' ,15, 'fontweight
362      ', 'bold')
363
364 end
365 fprintf('Tolerance: %.e\n',abstol(i))
366 fprintf('Van der Pol mu = %d\n')
367 varnames = { 'method' , 'N. Func' , 'N. Accept' , 'N. Reject' };
368 methodnames = { 'RK4' , 'ExplicitEuler' , 'ImplicitEuler' , 'DOPRI54' ,
369      'ESDIRK23' };
370 Table=table(methodnames' , [statRK4.nfun ,statEE.nfun ,statIE.nfun ,statD.
371      nfun ,info.nFun] '...
372      [statRK4.nAccept ,statEE.nAccept ,statIE.nAccept ,statD.nAccept ,info.
373      nAccept] ' , ...
374      [statRK4.nReject ,statEE.nReject ,statIE.nReject ,statD.nReject ,info.
375      nFail] ' , 'VariableNames' , varnames);
376 disp(Table)
377
378 fprintf('Tolerance: %.e\n',abstol(i))
379 fprintf('Van der Pol mu = 20\n')
380 Table2=table(methodnames' , [statRK42.nfun ,statEE2.nfun ,statIE2.nfun ,
381      statD2.nfun ,info2.nFun] '...
382      [statRK42.nAccept ,statEE2.nAccept ,statIE2.nAccept ,statD2.nAccept ,
383      info2.nAccept] ' , ...

```

```
370 [statRK42.nReject , statEE2.nReject , statIE2.nReject , statD2.nReject ,
371 info2.nFail ] ', 'VariableNames' , varnames );
372 disp( Table2 )
373 end
```

Listing A.22: Driver for problem 7 on the ESDIRK23 method.

Bibliography

- [1] U. Ascher and L. Petzold, *Computer Method for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics (SIAM), 1998.
- [2] D. Eberly, “Stability Analysis for Systems of Differential Equations.” [Online]. <https://www.geometrictools.com/Documentation/StabilityAnalysis.pdf>. [Accessed: May 20th 2022].
- [3] G. Fasshauer, “Chapter 4: Stiffness and Stability.” [Online]. Available: http://www.math.iit.edu/~fass/478578_Chapter_4.pdf. [Accessed: April 25th 2022].
- [4] J. Frank, “Chapter 10: Stability of Runge-Kutta Methods.” [Online]. Available: <https://webspace.science.uu.nl/~frank011/Classes/numwisk/>. [Accessed: April 25th 2022].
- [5] J. B. Jørgensen, “02685 Scientific Computing for Differential Equations.” [Online]. Available: <http://www.imm.dtu.dk/~jbjo/02685.html>. [Accessed: April 25th 2022].
- [6] M. Tsatsos, “Theoretical and numerical study of the van der pol equation,” 2006.
- [7] A. P. Keelers, “Wiener or brownian (motion) process,” 2021. [Online]. Available: <https://hpaulkeeler.com/wiener-or-brownian-motion-process>. [Accessed: April 30th 2022].
- [8] D. J. Higham, “An algorithmic introduction to numerical simulation of stochastic differential equations,” vol. 43, no. 3, p. 525–546, 2001.
- [9] T. Ritschel, “Numerical methods for solution of differential equations,” 2013. [Online]. Available: https://www2.imm.dtu.dk/pubdb/edoc/imm6607.pdf?fbclid=IwAR30x3Ds1EucGoIfdRJf1sJTWTl1VxI5B_btuCxilga1ibeWlMy18a8Z1-o. [Accessed: May 20th 2022].
- [10] E. Süli and D. Mayers, *An Introduction to Numerical Analysis*. Cambridge University Press, 2003.

- [11] M. R. Wahlgreen, E. Schroll-Fleischer, D. Boiroux, T. K. Ritschel, H. Wu, J. K. Huusom, and J. B. Jørgensen, “Nonlinear model predictive control for an exothermic reaction in an adiabatic cstr,” *IFAC-PapersOnLine*, vol. 53, no. 1, pp. 500–505, 2020. 6th Conference on Advances in Control and Optimization of Dynamical Systems ACODS 2020.
- [12] J. B. Jørgensen, M. R. Kristensen, and P. G. Thomsen, “A family of esdirk integration methods,” 2018.