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 Húnfjörð Jósepsdóttir (s212952)

Contents

CHAPTER 1

Test Equation for ODEs

In this chapter the test equation is considered:

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

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

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

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:

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

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:

$$e_n = x_n - x(t_n), \tag{1.4}$$

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:

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

$$e_n = x_n + exp(\lambda(t_n - t_0))x_0 \tag{1.6}$$

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 numberical 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.

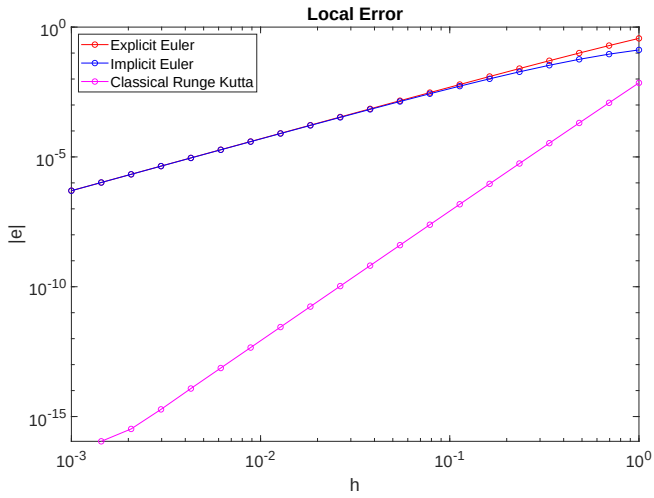


Figure 1.1: A log-log plot of the local truncation error, $|e|$, vs. step size, h, using explicit Euler, implicit Euler and the classical Runge-Kutta methods with a fixed step size on the test equation.

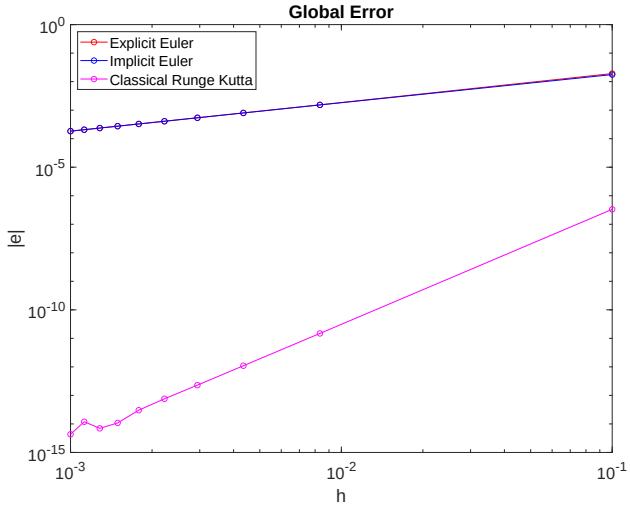


Figure 1.2: A log-log plot of global truncation error, $|e|$ vs. step size, h, using the explicit Euler, implicit Euler and the classical Runge-Kutta methods with a fixed step size on the test equation.

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 pertubations 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:

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

Given $\mathcal{R}e(\lambda) \leq 0$ then small perturbations in the initial condition causes only small perturbations in the solution and the problem is *stable*. On the other hand, if $\mathcal{R}e(\lambda) > 0$, 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 $\mathcal{R}e(\lambda) > 0$ 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:

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

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:

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

Now for a step of the implicit Euler one obtains:

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

The stability condition for the implicit Euler is therefore:

$$
\mathcal{D}_{implicitEuler} = \{z = \lambda h \in \mathcal{C} : \big|\frac{1}{1-z}\big| < 1\} \tag{1.10}
$$

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:

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

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

$$
\begin{aligned}
X &= x_n \mathbf{1} - zAY, \\
\Rightarrow X &= x_n (I - zA)^{-1} e,
\end{aligned} \tag{1.12}
$$

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

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

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

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

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:

$$
|R(z)| < 1, \;\; \forall z : \mathcal{R}e(z) < 0. \tag{1.15}
$$

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

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

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 $\lim_{z\to-\infty} |\mathcal{R}(z)| \to \infty$ so it is not L-stable, it's stability is dependent on the choice of the step size. On the other hand $\lim_{z\to-\infty} |\mathcal{R}(z)| = 0$ 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 satisfid as $\lim_{z\to-\infty} |\mathcal{R}(z)| \to \infty$. 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.

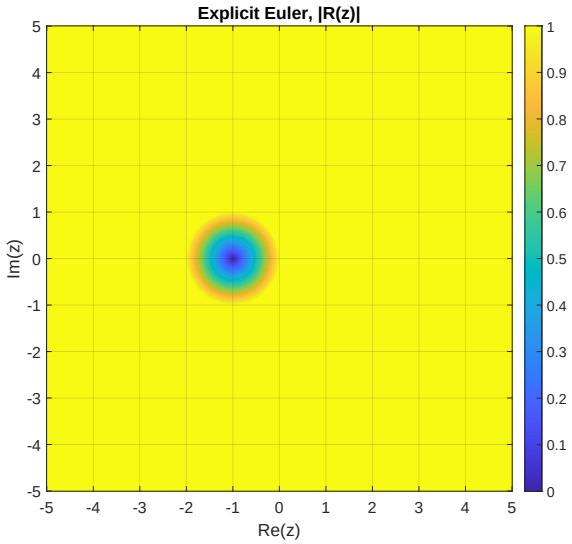


Figure 1.3: Stability region of the explicit Euler method.

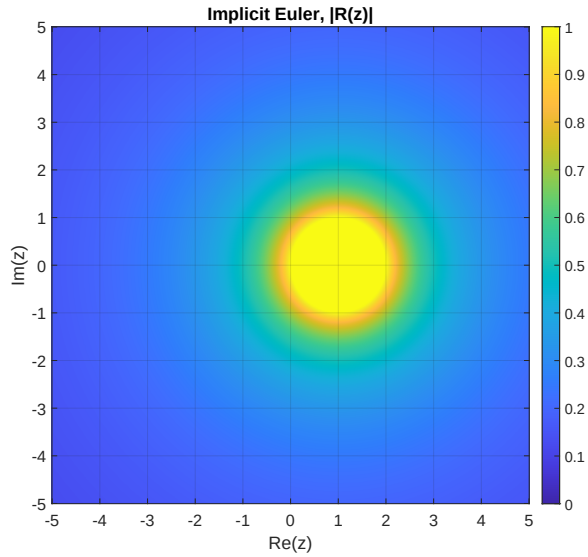


Figure 1.4: Stability region of the implicit Euler method.

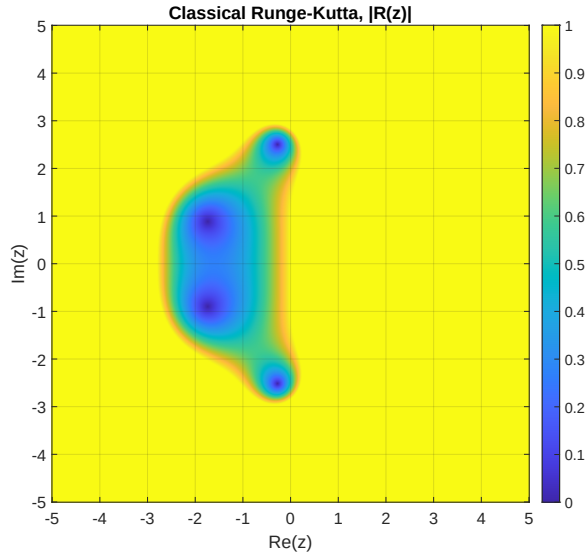


Figure 1.5: Stability region of the classical Runge-Kutta method.

CHAPTER 2

Explicit ODE solver

In this chapter the following initival value problem is considered:

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

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 it's 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:

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

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

$$x_0, x_1, ..., x_{N-1}, x_N,$$

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

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

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:

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

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

$$|x_k - \hat{x}_k| \leq TOL \tag{2.3}$$

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

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

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

Define error controller parameters
$\epsilon = 0.8$ `// Target`
$\mathcal{F}_{min} = 0.1$ `// Maximum decrease factor`
$\mathcal{F}_{max} = 5.0$ `// Maximum increase factor`
while *Final time t_f has not been reached* **do**
  if $t + h > t_f$ **then**
    $h = t_f - t$ `// Reduce step`
  end
  $f = F(t_0, x)$ `// Evaluate function at current point`
  while *Step is NOT accepted* **do**
    $x_{next} = x + hf$
    Step doubling
    $h_{temp} = \frac{1}{2}h$
    $t_{temp} = t + h_{temp}$
    $x_{temp} = x + h_{temp}f$ `// Take half a step`
    $f_{temp} = F(t_{temp}, x_{temp})$ `// Evaluate function at half-step`
    $\hat{x} = x_{temp} + h_{temp} * f_{temp}$
    Error estimation
    $e = \hat{x} - x_{next}$
    $r = max\{\frac{|e|}{max\{abstol, |\hat{x}|reltol\}}\}$
    if $r \leq 1.0$ **then**
      Step is accepted - update point
      $t := t + h$
      $x := \hat{x}$
    end
    Update step size using asymptotic step size controller
    $h := max\{\mathcal{F}_{min}, min\{\sqrt{\epsilon/r}, \mathcal{F}_{max}\}\} * h$
  end
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:

$$y''(t) = \mu(1 - y(t)^2)y'(t) - y(t). \tag{2.5}$$

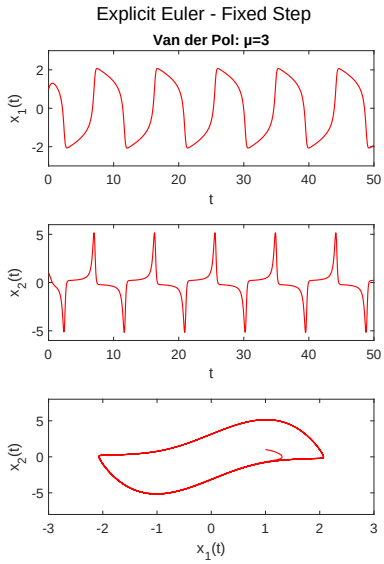
The problem can be converted to a system of first order differential equations:

$$\dot{x}_1(t) = x_2(t) \tag{2.6a}$$
$$\dot{x}_2(t) = \mu(1 - x_1(t)^2)x_2(t) - x_1(t), \tag{2.6b}$$

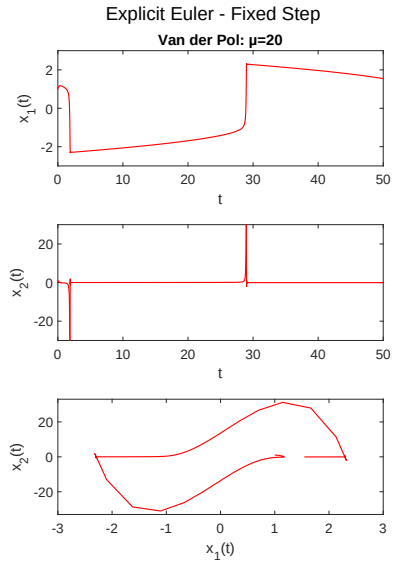
with $y(t) = x_1(t)$. The scalar parameter μ indicates the nonlinearity and strength of the dampening and the stiffness of the problem depends on this parameter. In the preceding subsections, the explicit Euler with fixed and adaptive step size described above are tested on the Van der Pol problem. The 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

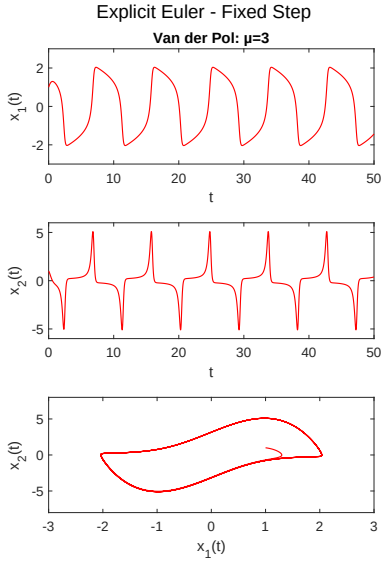
The results for the explicit Euler with fixed step size are visualized in Figure 2.1. As can be seen, more than $N = 3000$ steps are required for the stiff Van der Pol ($\mu = 20$) to avoid a large error. This is due to the fact that when μ is large it can cause rapid variations in the solution. The 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.



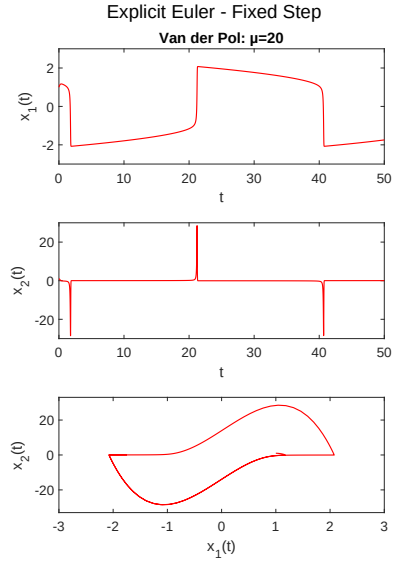
((a)) $h = 0.0167$.



((b)) $h = 0.0167$.



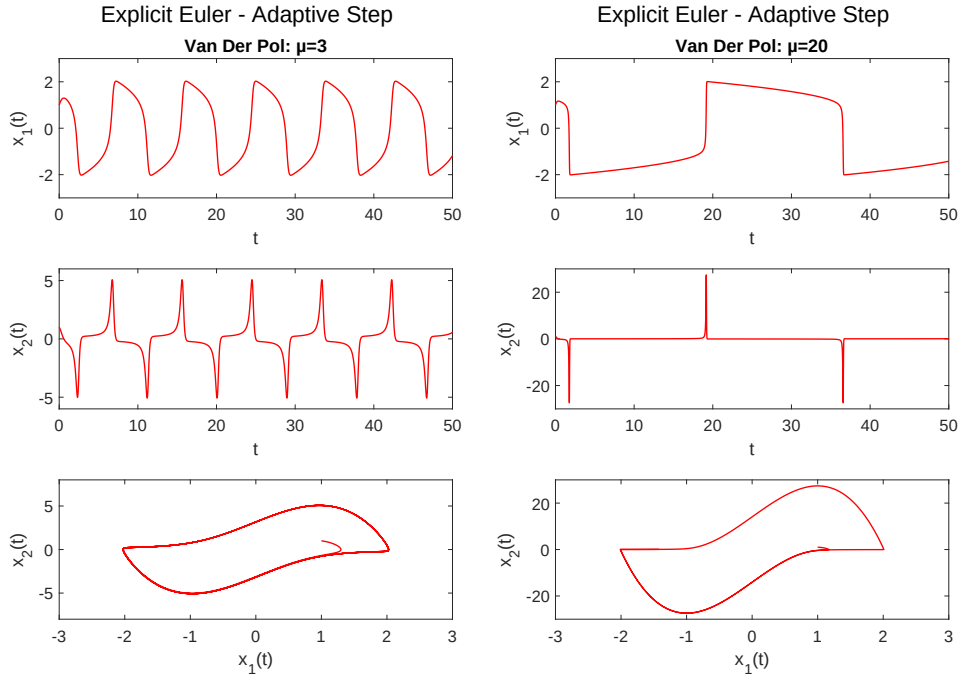
((c)) $h = 0.005$.



((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 $\mu = 20$.

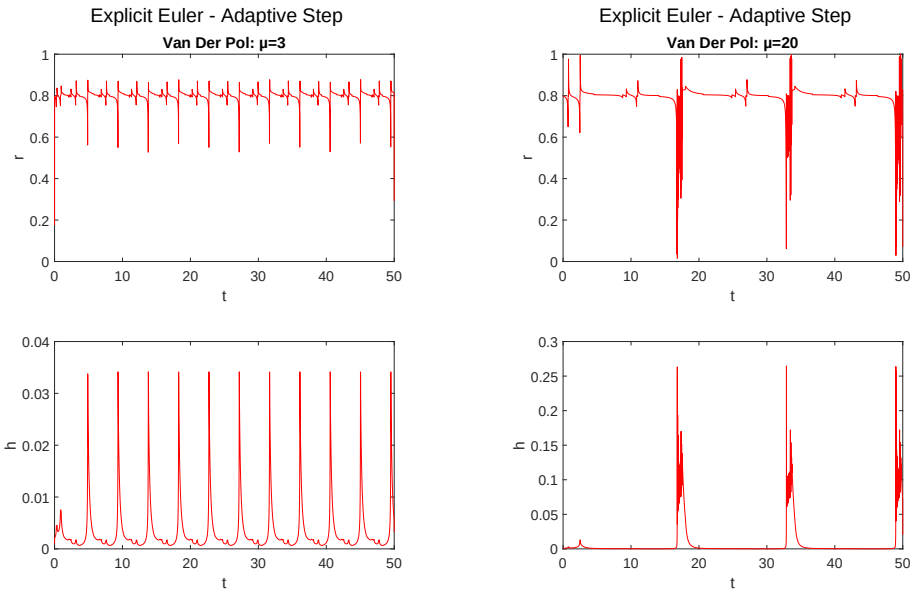
2.1.5 Explicit Euler with Adaptive Step Size and Error Estimation



((a)) ((b))

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.



((a)) **((b))**

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}.

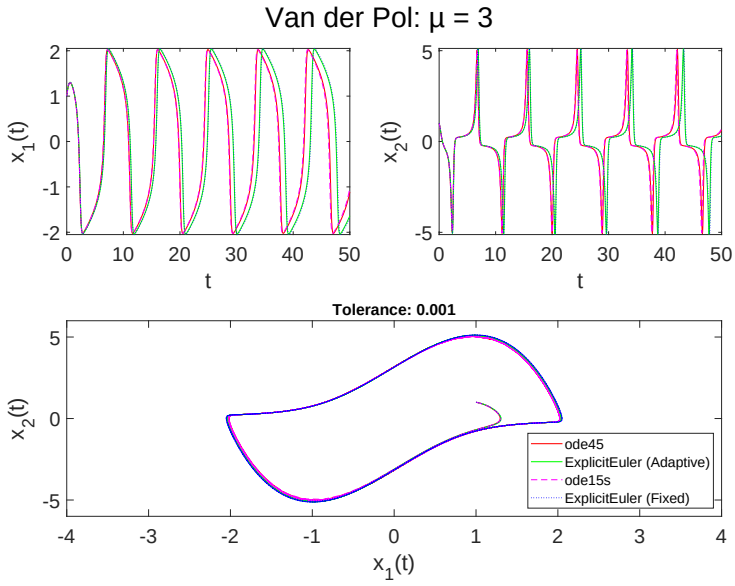
	Explicit Euler		**ode45**		**ode15s**	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
N. Function	3175	2142	1489	3751	1393	750
N. Steps	1658	1177	248	625	675	363
N. Accepted	1517	965	194	583	559	287
N. Rejected	141	212	54	42	116	76

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

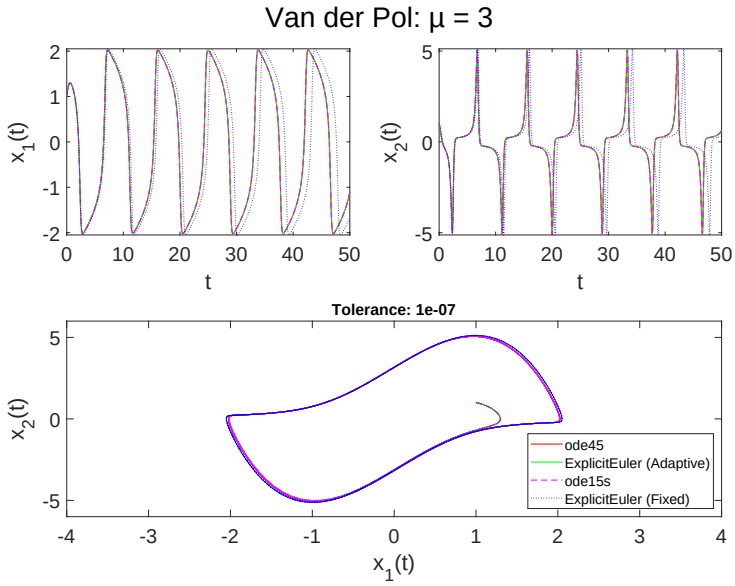
	Explicit Euler		**ode45**		**ode15s**	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
N. Function	300800	134550	6307	6769	4420	2325
N. Steps	150401	67273	683	1128	2659	1274
N. Accepted	150400	67273	656	1115	2505	1171
N. Rejected	1	2	27	13	154	103

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}.

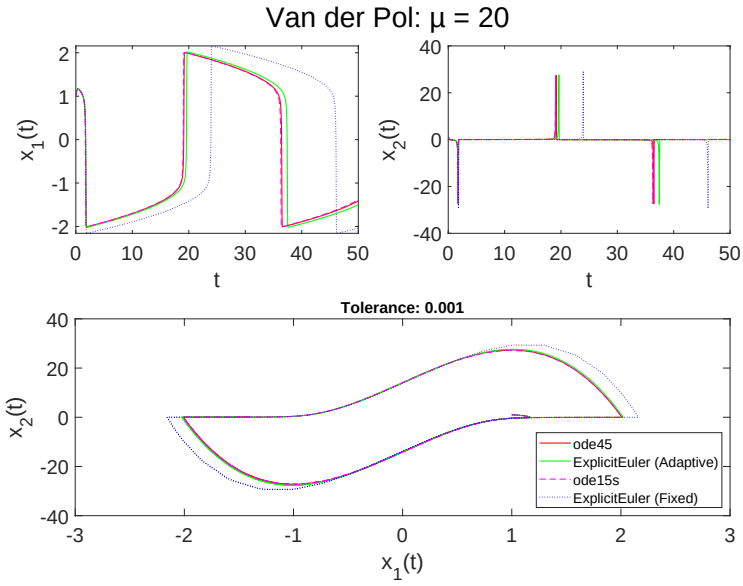


((a))

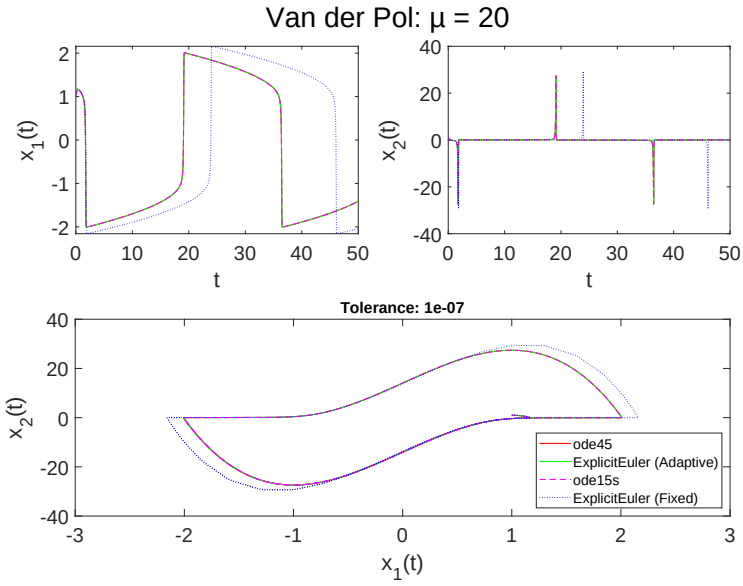


((b))

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



((a))



((b))

Figure 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:

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

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 numberical 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:

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

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

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

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

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

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

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

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

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

Newton's method is:

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

$$x_{k+1} = x_k + \nabla x \tag{3.8}$$

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:

$$x_{k+1} = x_k - \nabla x \tag{3.9}$$

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

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

Procedure *NewtonsMethodODE*
Data: t_i, x_i, x_{init}, step size h, objective function f, ϵ
Result: x
Set $t = t_i + h$
Set $x = x_{init}$
$[f, J] = F(t, x)$ `// Evaluate function at current point`
Set $x = x_{init}$
Set $R = x - f_i * h - x_i$ `// Initialize residual term`
while $||R||_2 > \epsilon$ **do**
  $\frac{\partial R}{\partial x} = I - J * h$
  $\nabla x = R / \frac{\partial R}{\partial x}$
  $x := x - \nabla x$
  $[f, J] = F(t, x)$ `// Evaluate function at current point`
  $R := x - f * h - x_i$ `// Update residual term`
end

3.2 The Impicit 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.

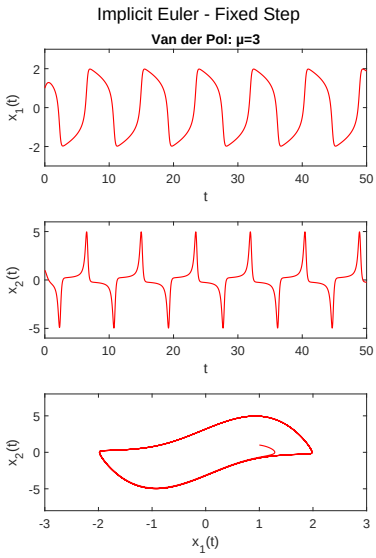
Data: t_0, t_f, x_0, initial step size h_0, objective function F, abstol,reltol
Result: t, x

Set $\epsilon = 0.8$ `// Tolerance`
Set $\mathcal{F}_{min} = 0.1$ `// Maximum decrease factor`
Set $\mathcal{F}_{max} = 5.0$ `// Maximum increase factor`
Set $h = h_0$
Set $x = x_0$
while *Final time t_f has not been reached* **do**
  if $t + h > t_f$ **then**
    $h = t_f - t$ `// Reduce step`
  end
  $f = F(t_i, x_i)$ `// Evaluate function at current point`
  while *Step is NOT accepted* **do**
    $x_{init} = x + f * h$
    $x_{next} = NewtonsMethodODE(t,x,x_{init},h,\mathrm{F},\epsilon)$ `// Approximate next step`
    Step doubling
    $h_{temp} = 0.5h$
    $t_{temp} = t + h_{temp}$
    $\hat{x}_{init} = x + fh_{temp}$ `// Take half a step`
    $x_{temp} = NewtonsMethodODE(t,x,\hat{x}_{init}, h_{temp},\mathrm{F},\epsilon)$
    $f_{temp} = F(t_{temp}, x_{temp})$ `// Evaluate function`
    $\hat{x} = x_{temp} + h_{temp} * f_{temp}$
    Error estimation
    $e = x_{next} - \hat{x}$
    $r = max\{\frac{|e|}{max\{abstol, |\hat{x}|reltol\}}\}$
    if $r \leq 1.0$ **then**
      Step is accepted - update point
      $t := t + h$
      $x := \hat{x}$
    end
    Update step size using asymptotic step size controller
    $h := max\{\mathcal{F}_{min}, min\{\sqrt{\epsilon/r}, \mathcal{F}_{max}\}\}h$
  end
end

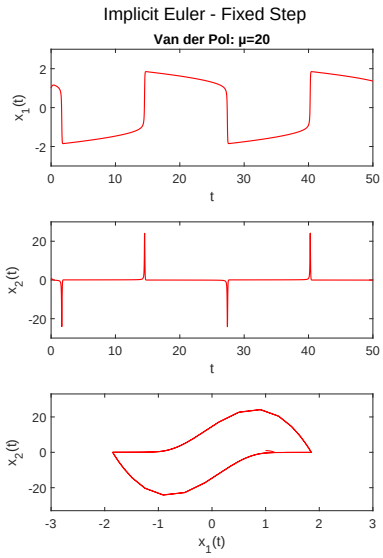
3.3 Testing the Implicit Euler Method on the Van der Pol Problem

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.

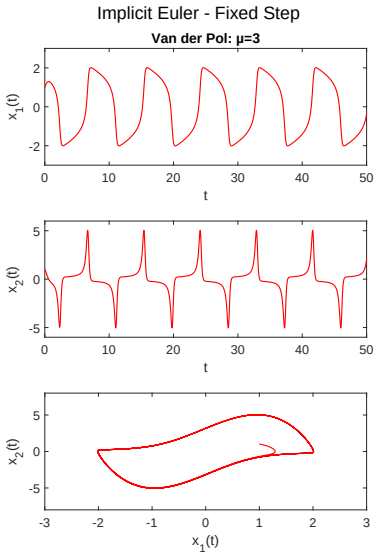
Figure 3.1 shows the results for using the implicit Euler with fixed step size on the Van der Pol problem in (2.6). Apparently, the implicit Euler does a better job of approximating the stiff system for smaller step size than the explicit Euler above, see Figure 3.1(b). According to Chapter 3 in the book by Ascher and Petzold [1] this makes sense as in general the implicit Euler requires fewer steps than the explicit for stiff problems. Additionaly, 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.



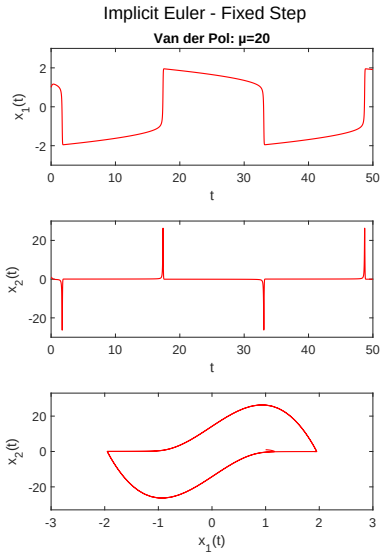
((a)) $h = 0.0167$.



((b)) $h = 0.0167$.



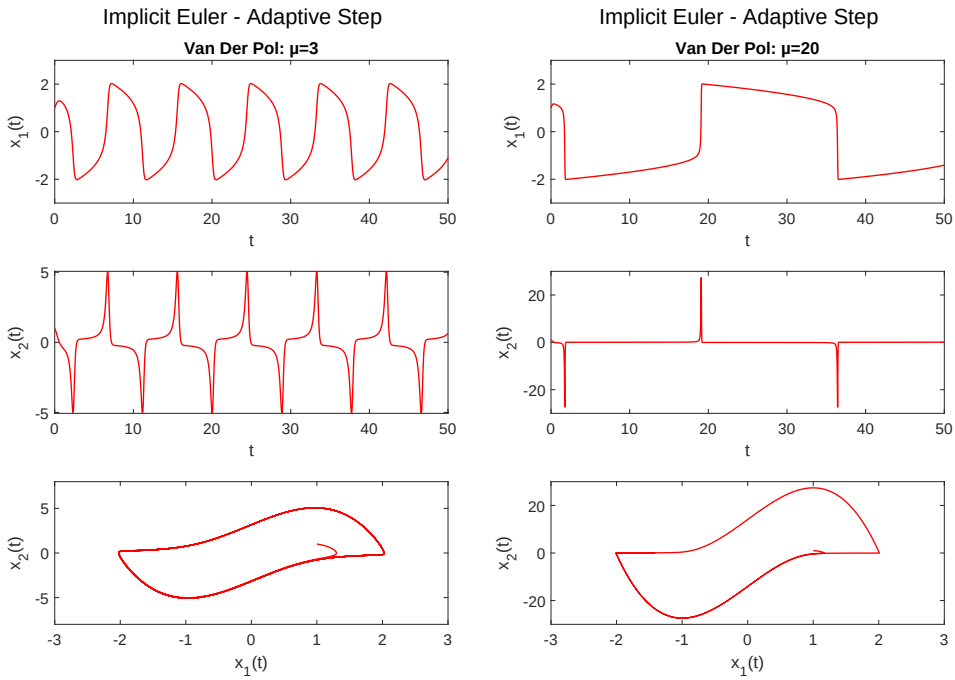
((c)) $h = 0.005$.



((d)) $h = 0.005$.

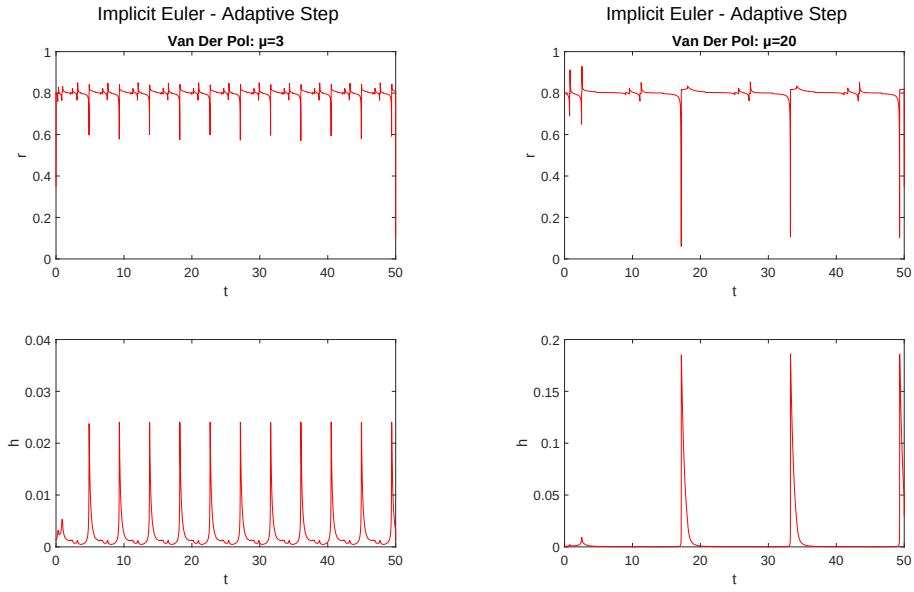
Figure 3.1: Implicit Euler with fixed step size, h, on the Van der Pol problem, non-stiff $\mu = 3$ and stiff $\mu = 20$.

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 $\mu = 20$.



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

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.

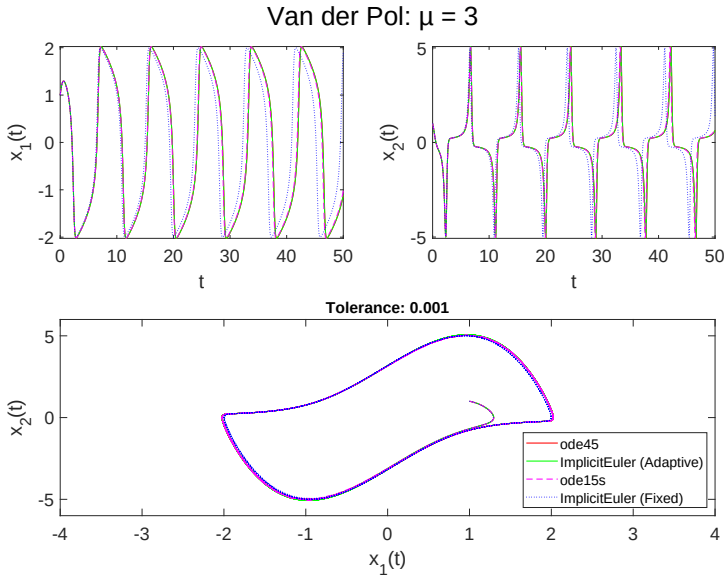
Table 3.1: Comparison of implicit Euler with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-3}.

	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

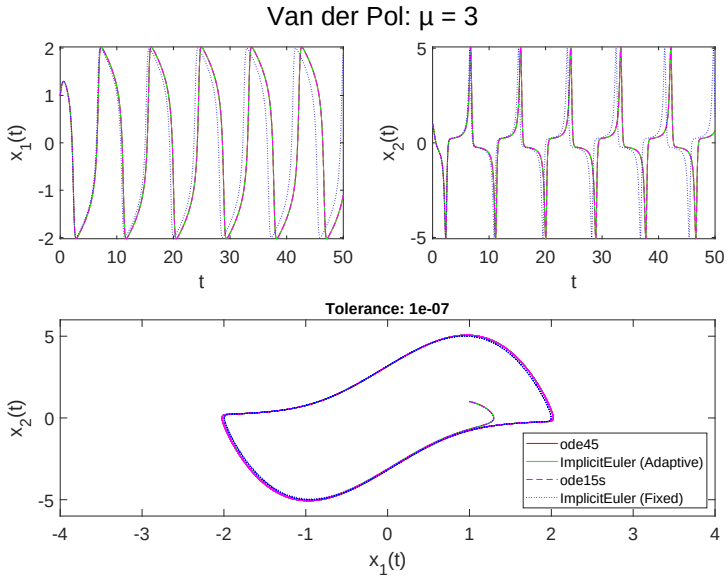
Table 3.2: Comparison of implicit Euler with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-7}.

	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.

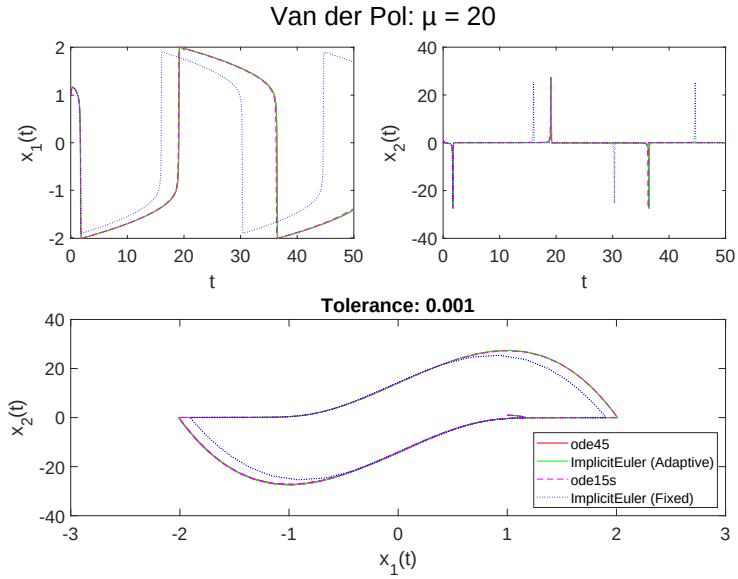


((a))

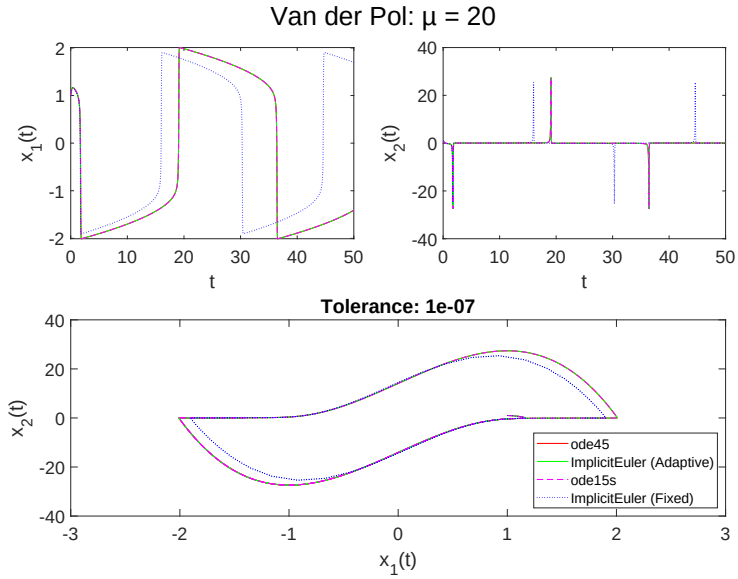


((b))

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:

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

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 $f : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_{p_f}} \mapsto \mathbb{R}^{n_x}$ and $g : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_{p_g}} \mapsto \mathbb{R}^{n_x \times n_\omega}$. 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:

$$\omega(t) = \int_0^t d\omega(s). \tag{4.2}$$

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:

$$\Delta t = \frac{T}{N}, \tag{4.3}$$

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:

$$\omega_k = \omega_{k-1} + d\omega_k, \quad j = 1, 2, ..., N, \tag{4.4}$$

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:

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

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

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

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

Set $x_1 = x_0$
Obtain number of time steps N from T
for *i from 1 to N-1* **do**
    $f_i = f(T_i, x_i)$ `// Evaluate function`
    $g_i = g(T_i, x_i)$
    $\nabla t = T_{i+1} - T_i$
    $\nabla \omega = \omega_{i+1} - \omega_i$
    $\psi = x_i + g\nabla\omega$
    Update x
    $x_{i+1} = \psi + f_i \nabla t$
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:

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

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

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

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

Obtain number of time steps N from T
Set $\epsilon = 10^{-8}$ `// Tolerance for Newton step`
Set $x_1 = x_0$
$f_1 = f(T_1, x_1)$ `// Evaluate function`
for *i from 1 to N-1* **do**
  $g_i = g(T_i, x_i)$ `// Evaluate function`
  $\nabla t = T_{i+1} - T_i$
  $\nabla \omega = \omega_{i+1} - \omega_i$
  $\psi = x_i + g \nabla \omega$
  $x_{init} = \psi + f_i \nabla t$
  $x_{i+1} = SDENewtonSolver(f, t_{i+1}, \nabla t, \psi, x_{init}, \epsilon)$ `// Obtain next step using Newton's method`
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
Set $x = x_{init}$
$[f, J] = F(t, x)$ `// Evaluate function at current point`
Set $R = x - f * \nabla t - \psi$ `// Initialize residual term`
while $||R||_\infty > \epsilon$ **do**
  $\frac{\partial R}{\partial x} = I - J * h$
  $\nabla x = R / \frac{\partial R}{\partial x}$
  $x := x - \nabla x$
  $[f, J] = F(t, x)$ `// Evaluate function at current point`
  $R := x - fh - x_i$ `// Update residual term`
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:

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

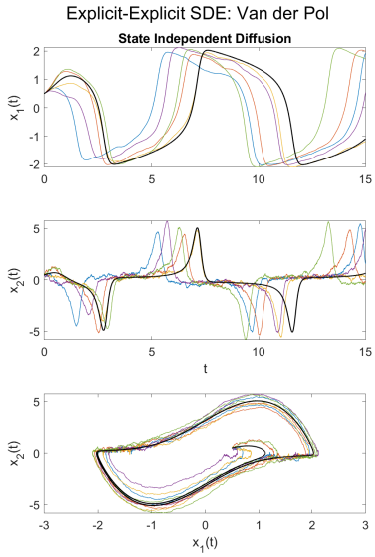
$$d\mathbf{x}_2(t) = [\mu(1-\mathbf{x}_1(t)^2)\mathbf{x}_2(t) - \mathbf{x}_1(t)]dt + \sigma d\boldsymbol{\omega}(t) \tag{4.10}$$

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

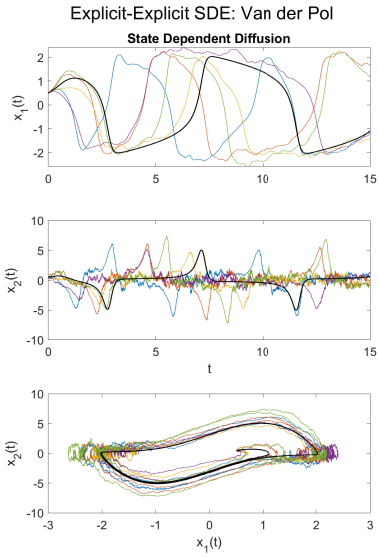
In the preceding sections, the methods derived above are tested on the SDE version of the Van der Pol problem. This was done for a non-stiff system with $\mu = 3$ and a stiff version $\mu = 20$ and large versus small step sizes compared, $N = [1000, 10000]$. The parameter $\sigma = 0.5$ and the initial 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

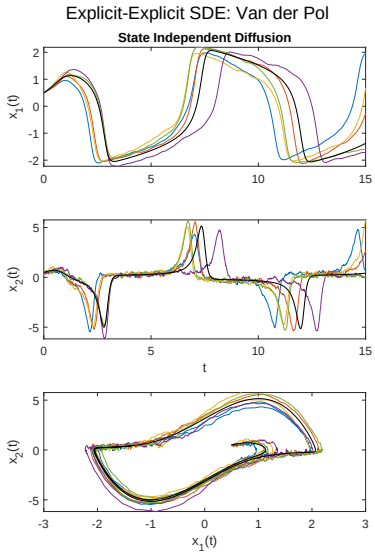
The results can be seen in the figures below. The black lines represent the drift term, corresponding to the ODE version of the Van der Pol, $\sigma = 0$. Figure 4.1 and Figure 4.3 show that the methods have good performance for the non-stiff system with both small and large steps. However, when it comes 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. Athough, 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.



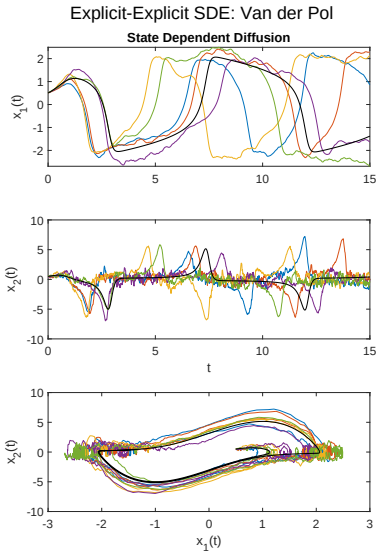
((a)) $N = 10000$.



((b)) $N = 10000$.

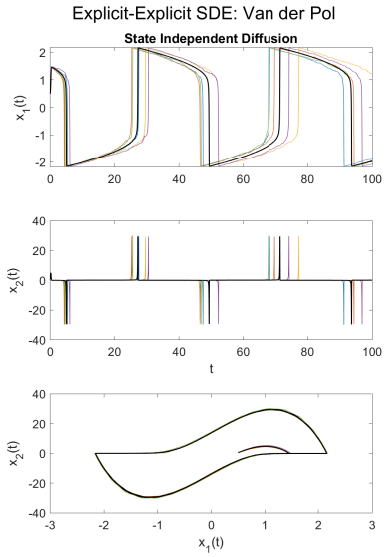


((c)) $N = 1000$.

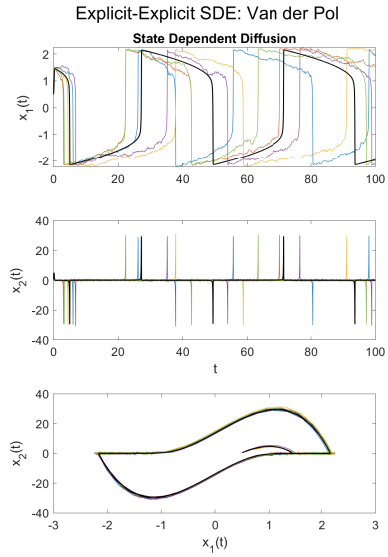


((d)) $N = 1000$.

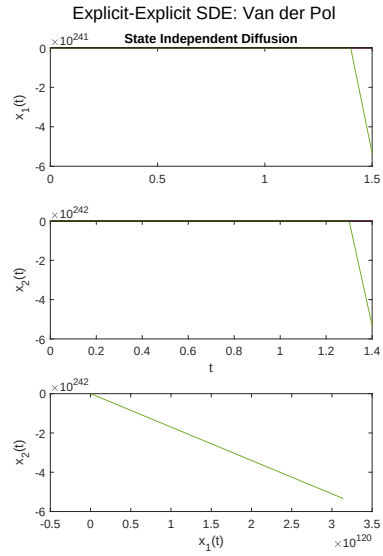
Figure 4.1: Explicit-Explicit Euler with fixed step size on an SDE version of the Van der Pol problem with $\mu = 3$ and $\sigma = 0.5$. The black lines show the drift term ($\sigma = 0$).



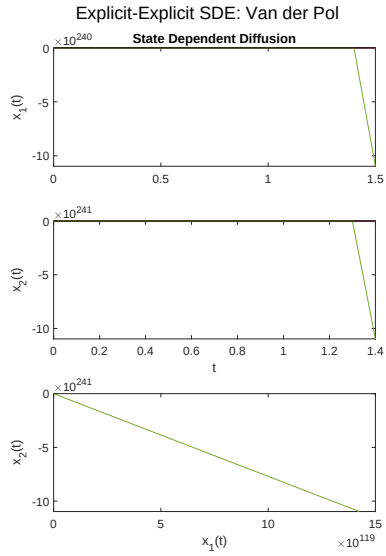
((a)) $N = 10000$.



((b)) $N = 10000$.

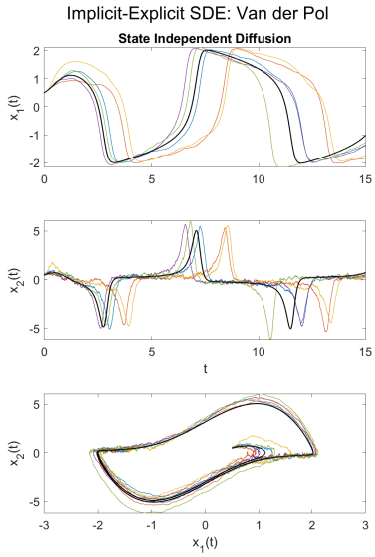


((c)) $N = 1000$.

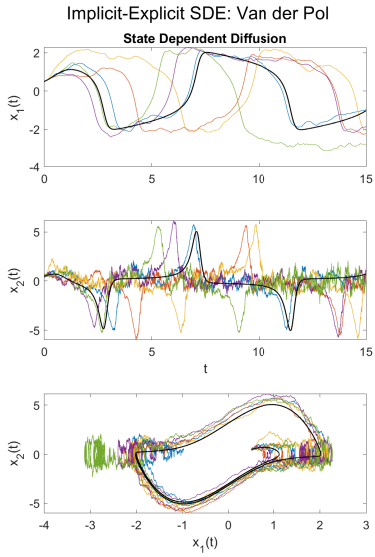


((d)) $N = 1000$.

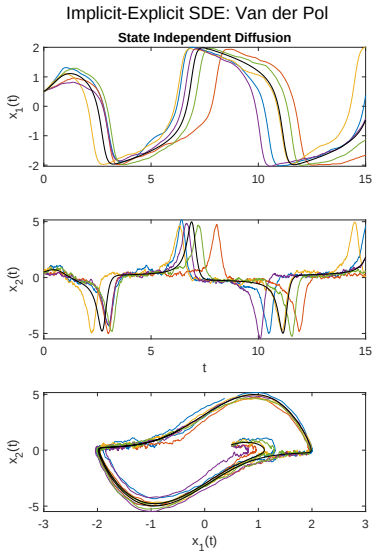
Figure 4.2: Explicit-Explicit Euler with fixed step size on an SDE version of the Van der Pol problem with $\mu = 20$ and $\sigma = 0.5$. The black lines show the drift term ($\sigma = 0$).



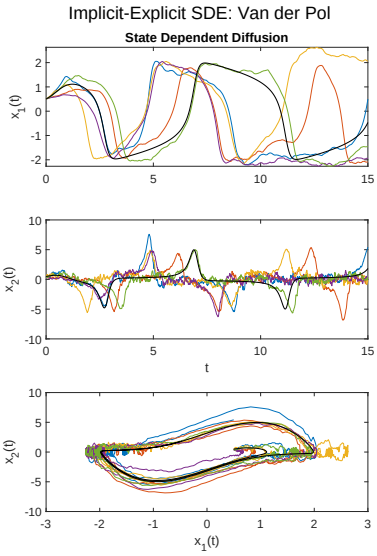
((a)) $N = 10000$.



((b)) $N = 10000$.

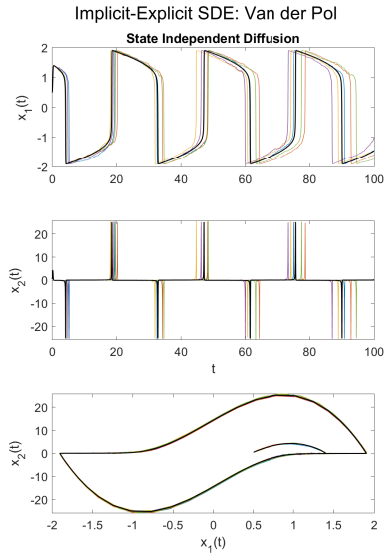


((c)) $N = 1000$.

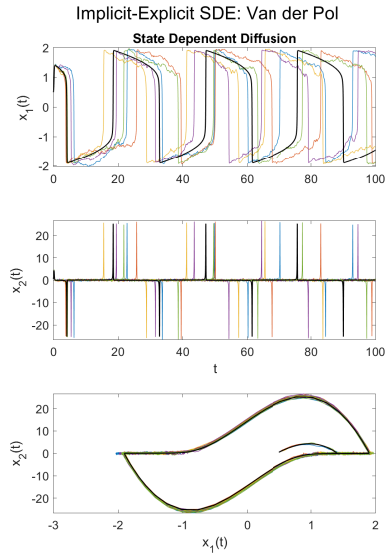


((d)) $N = 1000$.

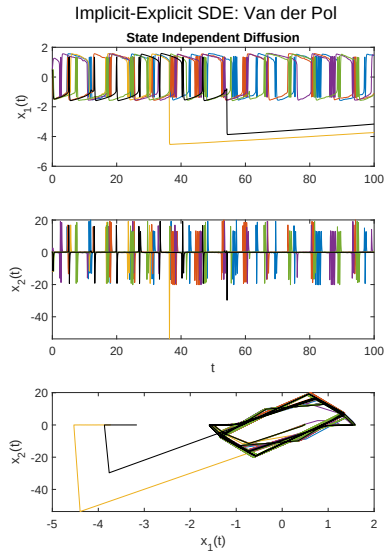
Figure 4.3: Implicit-Explicit Euler with fixed step size on an SDE version of the Van der Pol problem with $\mu = 3$ and $\sigma = 0.5$. The black lines show the drift term ($\sigma = 0$).



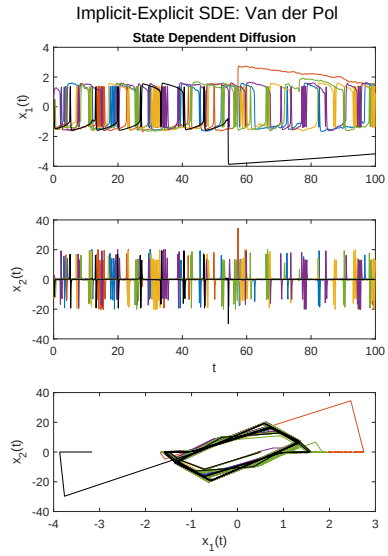
((a)) $N = 10000$.



((b)) $N = 10000$.



((c)) $N = 1000$.



((d)) $N = 1000$.

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:

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

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:

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

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

where a_{ij} are the internal coefficients and b_i for $i = 1, ..., s$ 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]:

$$c_i = \sum_{j=1}^{s} a_{ij}, \quad i = 1, ..., s \tag{5.3}$$

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

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

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

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:

$$X_1 = x_n \tag{5.6a}$$
$$X_2 = x_n + \frac{h}{2} f(X_1) \tag{5.6b}$$
$$X_3 = x_n + \frac{h}{2} f(X_2) \tag{5.6c}$$
$$X_4 = x_n + h f(X_3) \tag{5.6d}$$
$$x_{n+1} = x_n + h\big(\frac{1}{6} f(X_1) + \frac{1}{3} f(X_2) + \frac{1}{3} f(X_3) + \frac{1}{6} f(X_4)\big). \tag{5.6e}$$

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

Set $N = \frac{t_f - t_0}{h}$
Set $x = x_0$
for *n=1...N* **do**
  Stage 1
  Set $T_1 = t_0$
  Set $X_1 = x_0$
  $F_1 = f(T_1, X_1)$ `// Evaluate function at given point`
  Stage 2-4
  for *i=2,...,4* **do**
    $X_i = x + h\sum_{j=1}^{i-1} a_{ij}F_j$ `//` a_{ij} `from the A matrix`
    $T_i = t + hc_i$
    $F_i = f(T_i, X_i)$
  end
  Compute next step
  $t := t + h$
  $x := x + h\sum_{i=1}^{4} b_iF_i$ `//` b_i `from the b vector`
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]:

$$\hat{T}_i^{n+\frac{1}{2}} = t_n + c_i\frac{h}{2}$$

$$\hat{X}_i^{n+\frac{1}{2}} = x_n + \frac{h}{2}\sum_{j=1}^{i-1} a_{ij}f(\hat{T}_j^{n+\frac{1}{2}}, \hat{X}_j^{n+\frac{1}{2}})$$

$$\hat{x}_{n+\frac{1}{2}} = x_n + \frac{h}{2}\sum_{j=1}^{i-1} b_if(\hat{T}_j^{n+\frac{1}{2}}, \hat{X}_j^{n+\frac{1}{2}})$$

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

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.

Procedure: *ClassicalRungeKuttaStep*
Data: A, b, c from Butcher tableau, x, t, objective function f, step size h
Result: t, x
Stage 1
Set $T_1 = t$
Set $X_1 = x$
$F_1 = f(T_1, X_1)$ `// Evaluate function at given point`
Stage 2-4
for *i=2,...,4* **do**
  $X_i = x + h \sum_{j=1}^{i-1} a_{ij} F_j$ `//` a_{ij} `from the A matrix`
  $T_i = t + hc_i$
  $F_i = f(T_i, X_i)$
end
Compute next step
$t := t + h$
$x := x + h \sum_{i=1}^{4} b_i F_i$ `//` 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

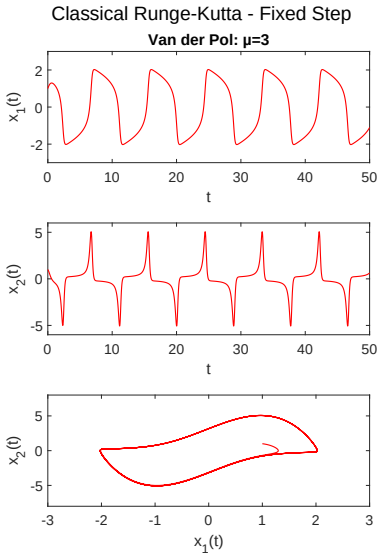
Set $\epsilon = 0.8$ `// Tolerance`
Set $\mathcal{F}_{min} = 0.1$ `// Maximum decrease factor`
Set $\mathcal{F}_{max} = 5.0$ `// Maximum increase factor`
Set $h = h_0$
Set $t = t_0$
Set $x = x_0$
while $t < t_f$ **do**
  if $t + h > t_f$ **then**
    $h = t_f - t$
  end
  f=F(t,x) `// Evaluate function at current point`
  while *Step is NOT accepted* **do**
    x_{next} = *ClassicalRungeKuttaStep*(A,b,c,t,x,F,h) `// Take full step`
    $h_{temp} = \frac{1}{2}h$
    $[t_{temp}, x_{temp}]$= *ClassicalRungeKuttaStep*(A,b,c,t,x,F,h_{temp}) `// Take half step`
    $f_{temp} = F(t_{temp}, x_{temp})$ `// Evaluate function`
    $[\hat{t}, \hat{x}]$ = *ClassicalRungeKuttaStep*(A,b,c,t_{temp},x_{temp},F,h_{temp})
    $e = x_{next} - \hat{x}$
    $r = max\{\frac{|e|}{max\{abstol, |\hat{x}| reltol\}}\}$
    if $r \leq 1.0$ **then**
      Step is accepted - update point
      $t := t + h$
      $x := \hat{x}$
    end
    Update step size using asymptotic step size controller
    $h := max\{\mathcal{F}_{min}, min\{\sqrt{\epsilon / r}, \mathcal{F}_{max}\}\}h$
  end
end

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

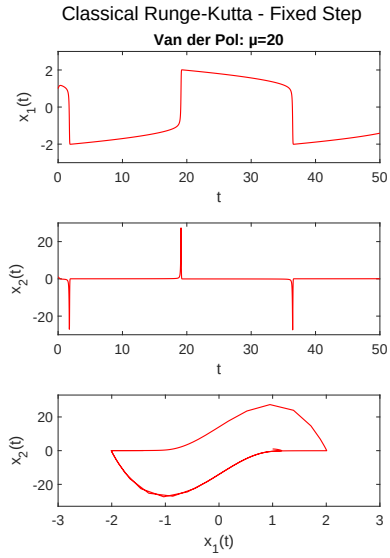
In this section, the classical Runge-Kutta methods derived above are tested on the Van der Pol problem given in (2.6) with initial point $x_0 = [1, 1]^T$. This was done for both stiff and non-stiff versions of the system and for a time span of $t = [0, 50]$. The driver for performing the tests can be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

5.3.1 Classical Runge-Kutta with Fixed Step Size

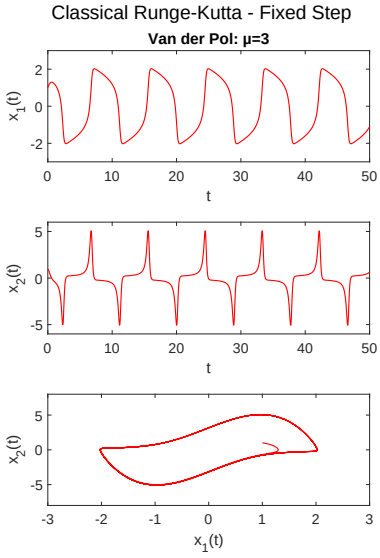
The classical Runge-Kutta with fixed step size was tested on the Van der Pol problem with a step size of $N = 3000$ and $N = 10000$ steps, with the same setup as in previous chapters. The results can be seen in Figure 5.1. The phase plot in Figure 5.1(b) is quite smooth with some sharp edges at times. 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 apparant difference between using smaller or larger steps in the results and the fluctuations are nice and smooth.



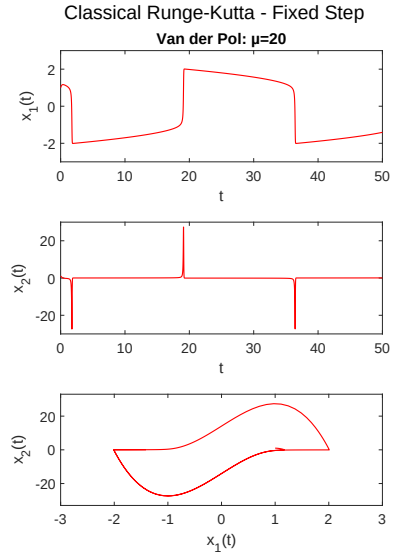
((a)) $h = 0.0167$.



((b)) $h = 0.0167$.



((c)) $h = 0.005$.

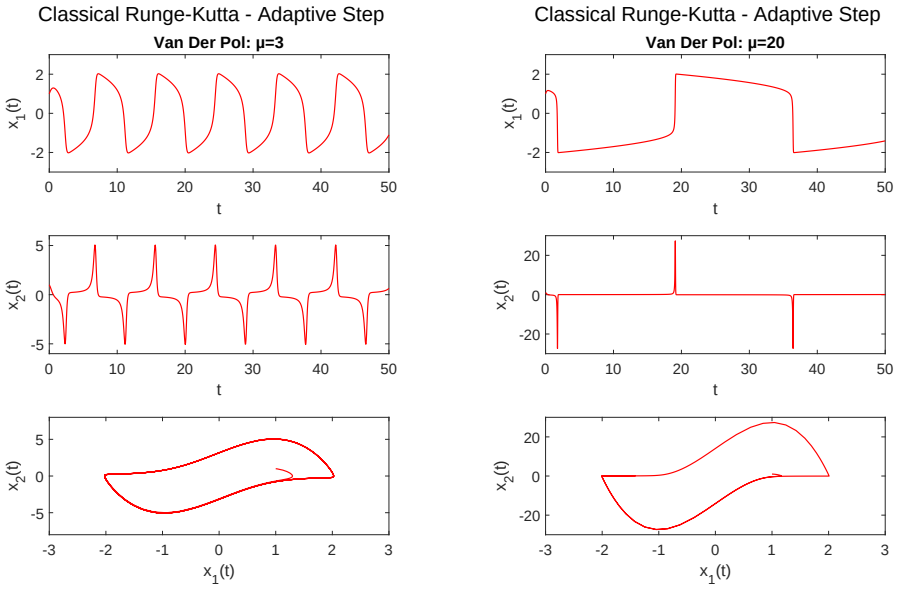


((d)) $h = 0.005$.

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

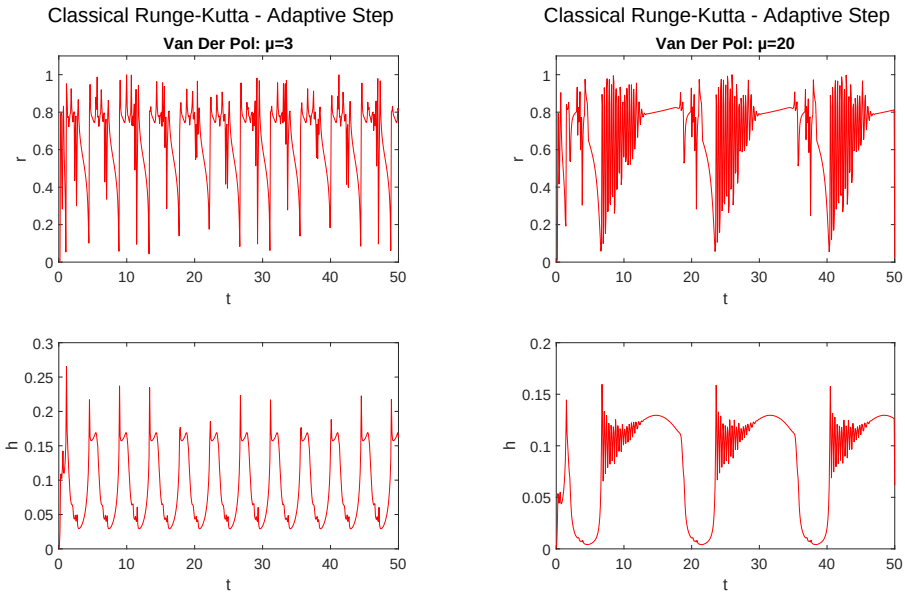
5.3.2 Classical Runge-Kutta with Adaptive Step Size

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 erratice behaviour. To conclude, the classical Runge-Kutta performs the when the problem is non-stiff and for stiff systems an implicit method would be preferable.



((a)) **((b))**

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.

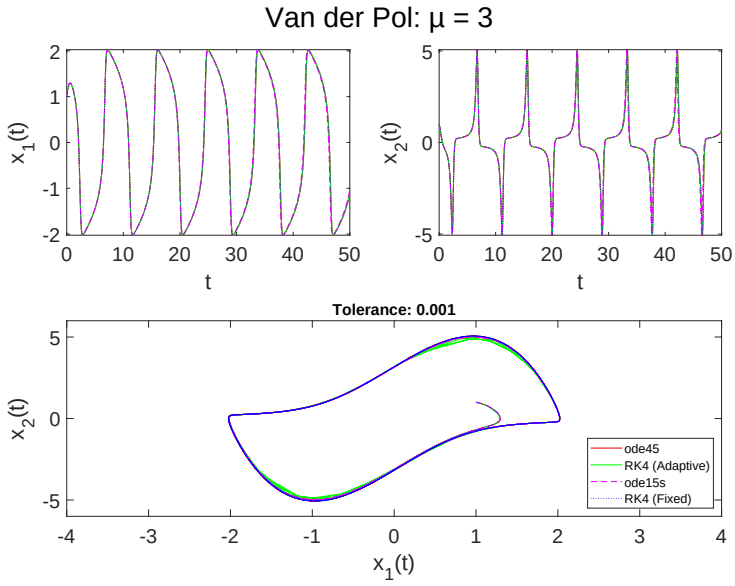
Table 5.2: Comparison of classical Runge-Kutta with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-3}.

	Runge-Kutta		**ode45**		**ode15s**	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
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

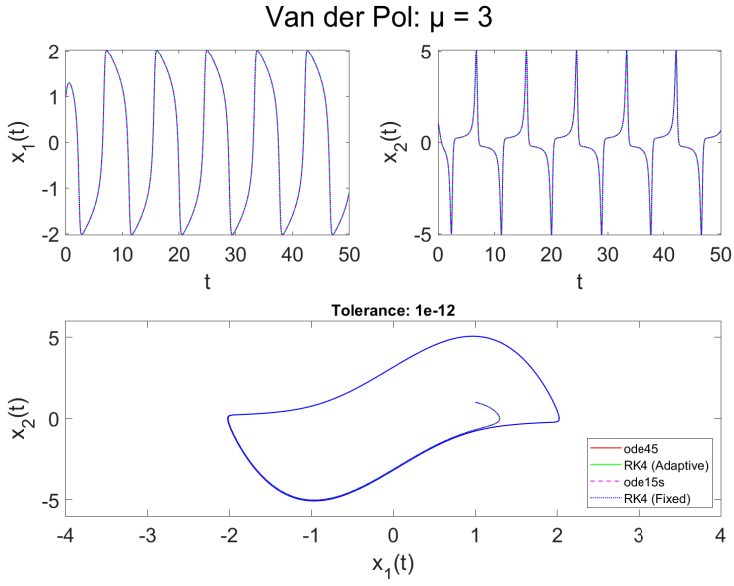
Table 5.3: Comparison of classical Runge-Kutta with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-12}.

	Runge-Kutta		**ode45**		**ode15s**	
	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
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.

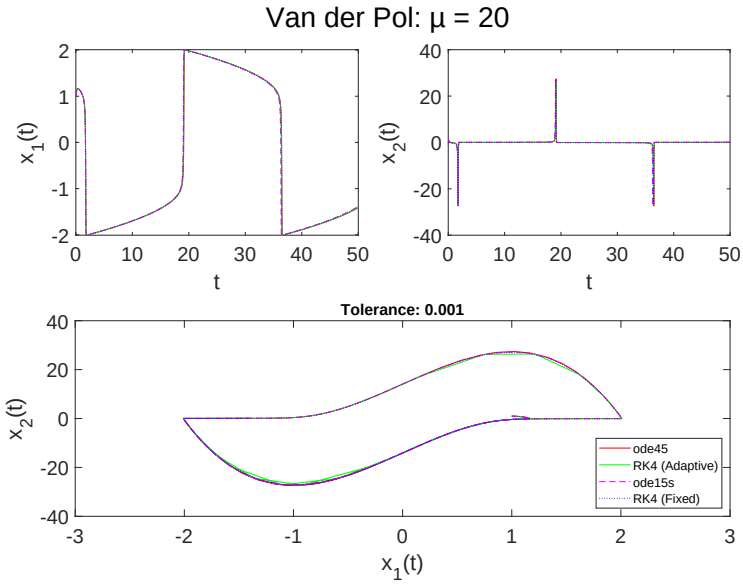


((a))

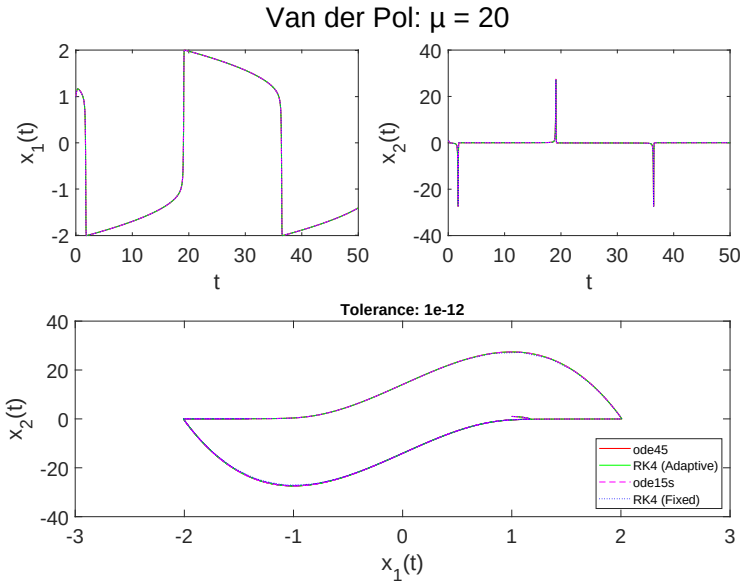


((b))

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



((a))



((b))

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:

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

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. Is 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 tableu.

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]:

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

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:

$$\begin{aligned} e &= \hat{x}_{n+1} - x_{n+1} && (6.3)\\ &= \big(x_n + h\sum_{j=1}^{r} b_j f(T_j, X_j)\big) - \big(x_n + h\sum_{j=1}^{r} \hat{b}_j f(T_j, X_j)\big) && (6.4)\\ &= h\sum_{j=1}^{r} (b_j - \hat{b}_j) f(T_j, X_j) && (6.5)\\ &= h\sum_{j=1}^{r} d_j f(T_j, X_j) && (6.6) \end{aligned}$$

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:

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

The full procedure for the DOPRI54 method with embedded error estimation and apadtive 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 it's 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

Set $s = 7$ // Number of stages in method
Set $\epsilon = 0.8$ // Tolerance
Set $\mathcal{F}_{min} = 0.1$ // Maximum decrease factor
Set $\mathcal{F}_{max} = 5.0$ // Maximum increase factor
Set $h = h_0$
Set $t = t_0$
Set $x = x_0$
while $t < t_f$ **do**
  if $t + h > t_f$ **then**
    $h = t_f - t$
  end
  f=F(t,x) // Evaluate function at current point
  while *Step is NOT accepted* **do**
    Take step of size h with embedded error estimation
    Set $T_1 = t$
    Set $X_1 = x$
    Set $f_1 = f$
    for $i = 2, .., s$ **do**
      $X_i = x + h \sum_{j=1}^{i-1} a_{ij} f_j$ // a_{ij} from the A matrix
      $T_i = t + hc_i$
      $f_i = F(T_i, X_i)$
    end
    Approximate next step and compute error
    $e := h \sum_{j=1}^{s} d_j f_j$
    $\hat{t} := t + c_7 h$
    $\hat{x} := X_7$
    Error estimation
    $r = max\{\frac{|e|}{max\{abstol, |\hat{x}| reltol\}}\}$
    if $r \leq 1.0$ **then**
      Step is accepted - update point
      $t := \hat{t} + h$
      $x := \hat{x}$
    end
    Update step size using asymptotic step size controller
    $h := max\{\mathcal{F}_{min}, min\{\left(\frac{\epsilon}{r_{n+1}}\right)^{1/5}, \mathcal{F}_{max}\}\}h$
  end
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.

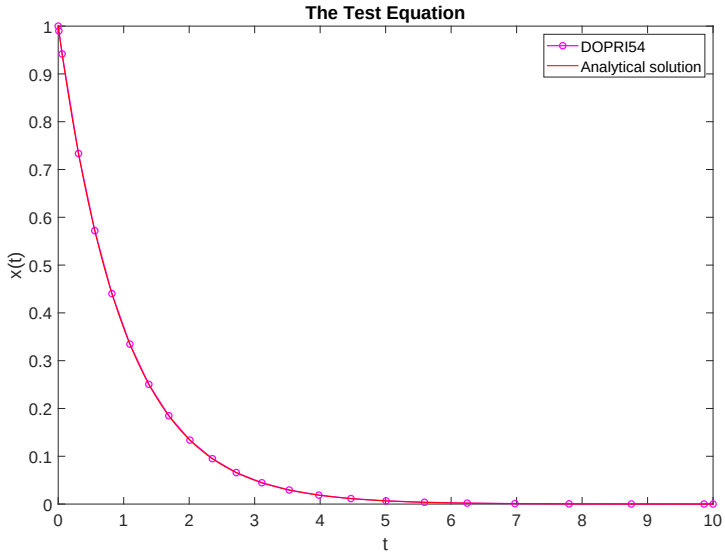


Figure 6.1: DOPRI54 solution to the test equation vs. the analytical solution.

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 presicion 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.

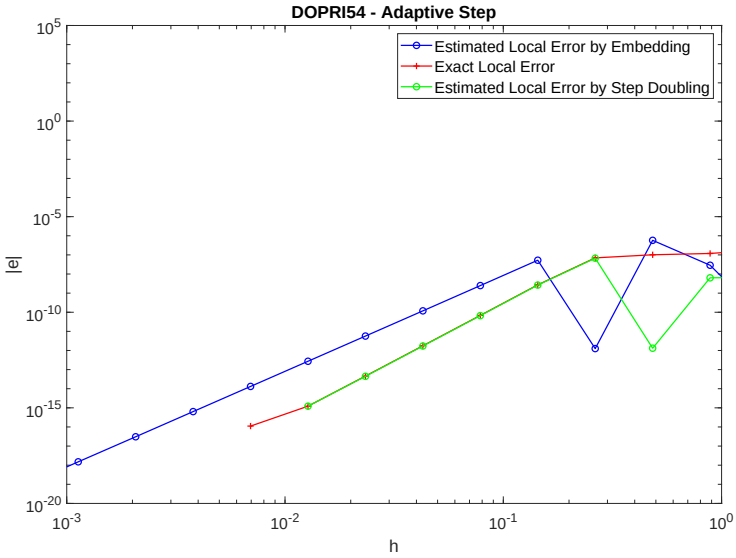


Figure 6.2: Log-log plot of the exact vs estimated local error of the DOPRI54 method.

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.

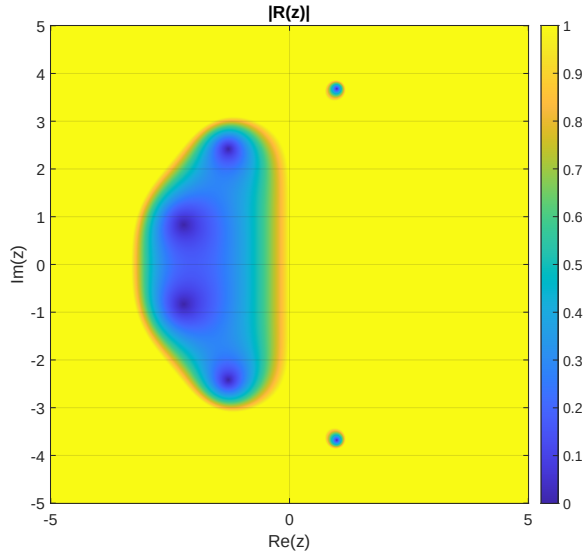
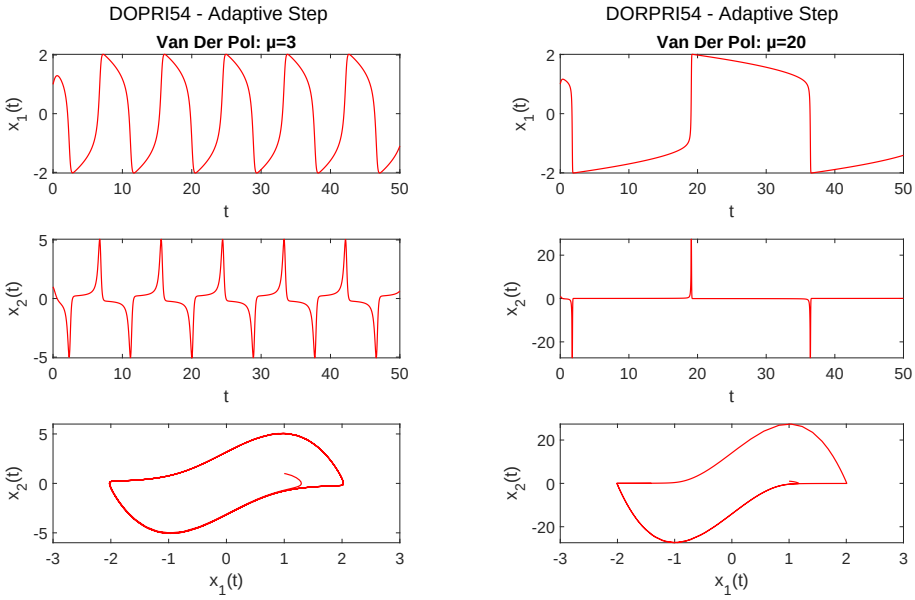


Figure 6.3: The stability region of the DOPRI54 method.

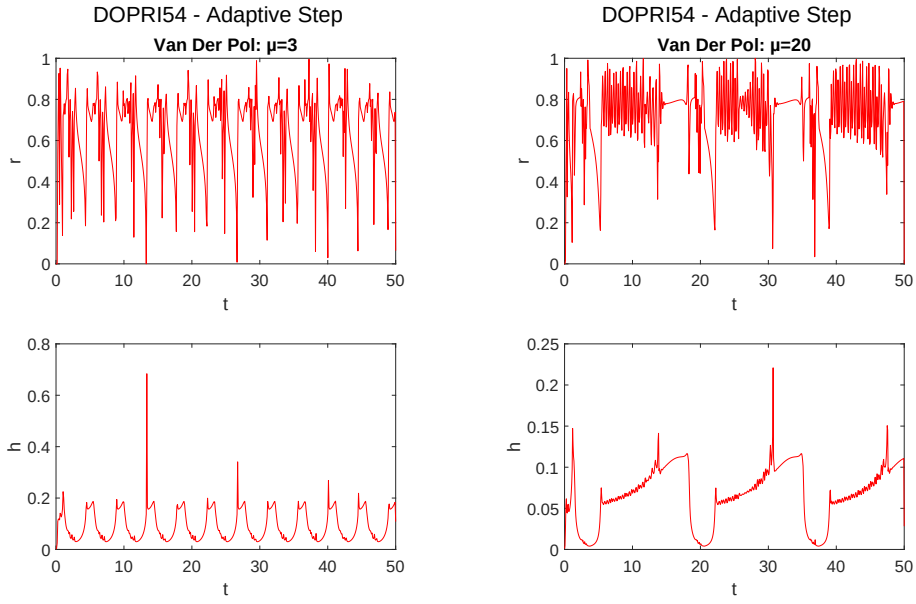
6.1.3 The Van der Pol Problem

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 aformentioned method.



((a)) **((b))**

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



((a)) **((b))**

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:

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

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

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

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:

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

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

$$\tag{6.9c}$$

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

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

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.

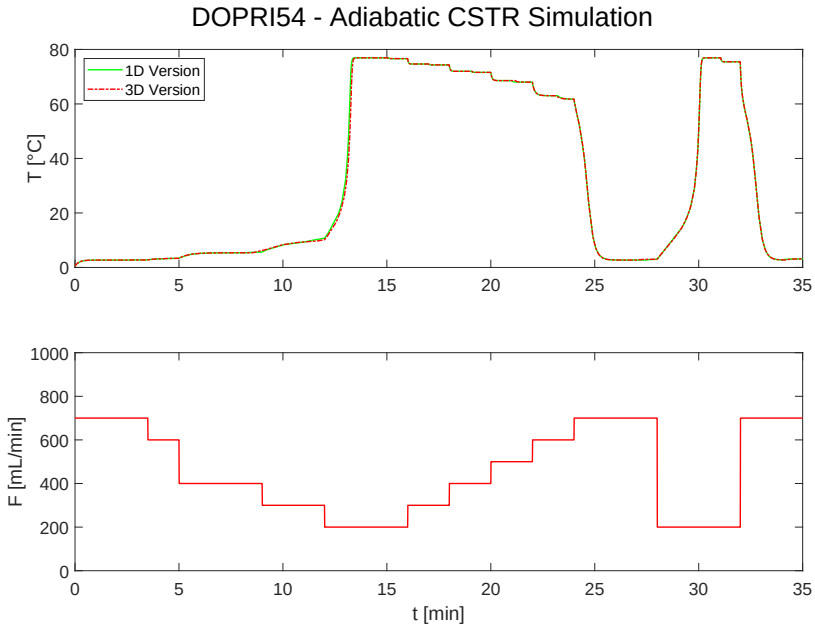


Figure 6.6: Simulation of adibiatic CSTR using the DOPRI54 method, 1D vs. 3D version.

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.

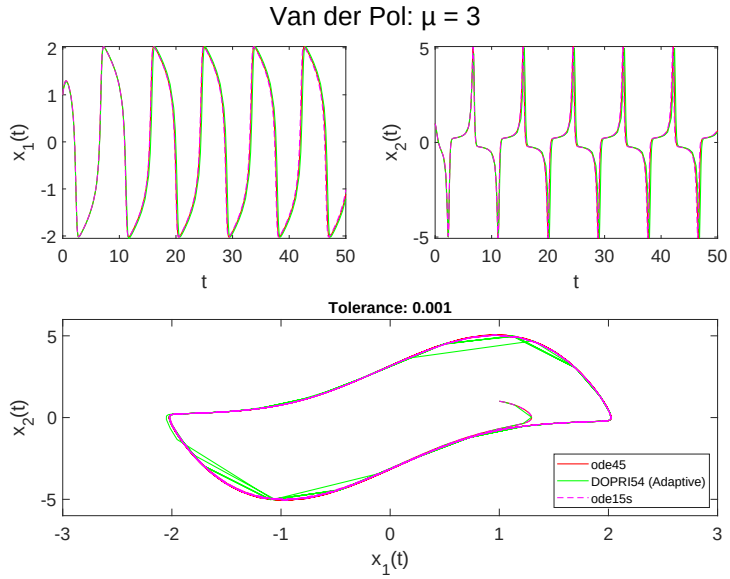
Table 6.2: Comparison of DOPRI54 with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-3}.

	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

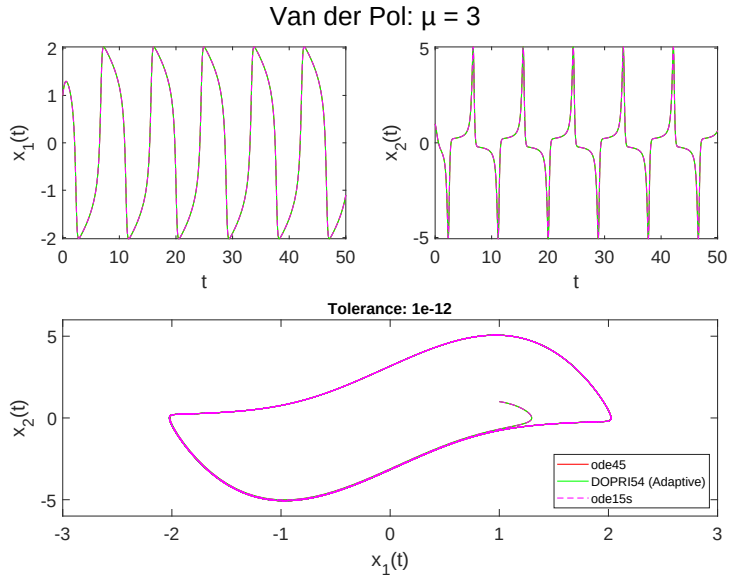
Table 6.3: Comparison of DOPRI54 with adaptive step size, *ode45* and *ode15s*. Absolute and relative tolerances at 10^{-12}.

	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.

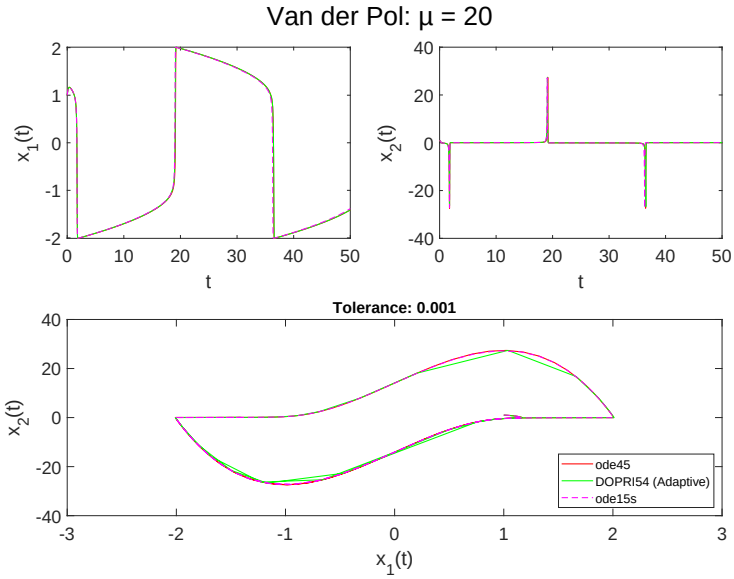


((a))

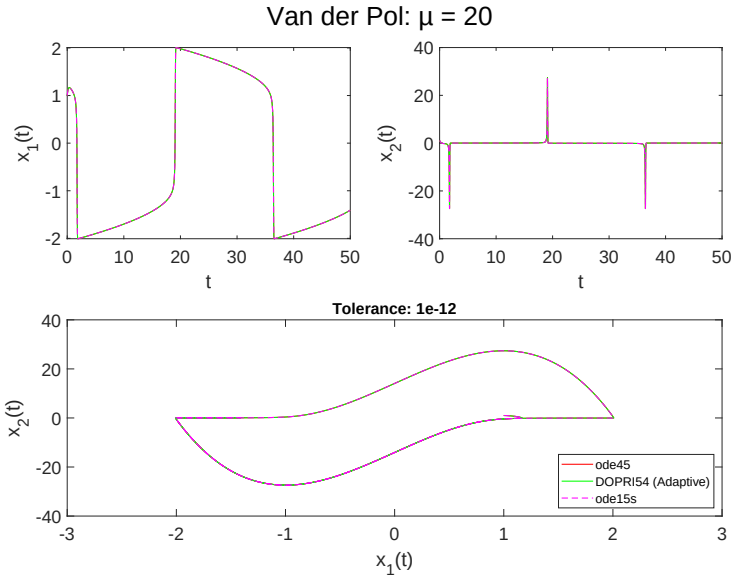


((b))

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



((a))



((b))

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 reaused, which saves computational cost. In the following sections, the ESDIRK23 method will be derived and it's performance and stability assessed. The driver for this chapter can be found in Appendix A.2.7.

7.1 ESDIRK23

ESDIR23 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:

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

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:

$$a_{21}b_2 - b_1\gamma = 0. \tag{7.2}$$

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:

$$c_2 = a_{21} + \gamma, \tag{7.3a}$$
$$1 = b_1 + b_2 + \gamma. \tag{7.3b}$$

The order conditions are:

$$1st\ order: \quad b_1 + b_2 + \gamma = 1, \tag{7.4a}$$
$$2nd\ order: \quad b_2 c_2 + \gamma = \frac{1}{2}. \tag{7.4b}$$

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

$$a_{21} + \gamma = c_2, \tag{7.5a}$$
$$\gamma c_2 = \frac{1}{2} c_2, \tag{7.5b}$$

which are equicalent 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:

$$1st\ order: \quad \hat{b}_1 + \hat{b}_3 + \hat{b}_3 = 1, \tag{7.6a}$$
$$2nd\ order: \quad \hat{b}_2 c_2 + \hat{b}_3 = \frac{1}{2}, \tag{7.6b}$$
$$3rd\ order: \quad \hat{b}_2 c_2^2 + \hat{b}_3 = \frac{1}{3}, \tag{7.6c}$$
$$\hat{b}_2(c_2\gamma) + \hat{b}_3(c_2 b_2 + \gamma) = \frac{1}{6}. \tag{7.6d}$$

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 $\lim_{z\to-\infty} |\mathcal{R}(z)| = 0$ and the two conditions for L-stability are therefore satisfied. To summaries, the method is both A-stable and L-stable.

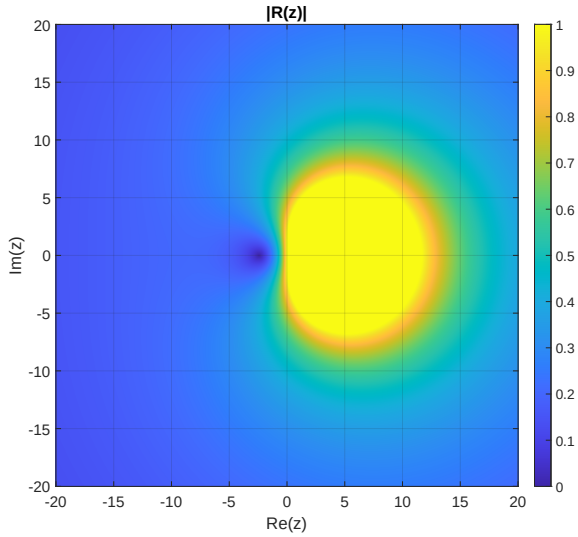


Figure 7.1: The stability region of the ESDIRK23 method.

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:

$$T_1 = t_n \qquad X_1 = x_n \tag{7.7}$$

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

$$T_3 = t_n + h \qquad X_3 = x_n + ha_{31}f(X_1, T_1) + ha_{32}f(T_2, X_2) + h\gamma f(T_3, X_3) \tag{7.9}$$

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:

$$M \approx I - h\gamma J(t_n, x_n), \tag{7.10}$$

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:

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

where:

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

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

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

$$X_2 = x_n + c_2 hf(t_n, x_n) = x_n + c_2 hf_1. \tag{7.14}$$

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

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

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:

$$x_{n+1} = X_3, \tag{7.16}$$

$$t_{n+1} = T_3. \tag{7.17}$$

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

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

The measure of the error estimate is calculated as:

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

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

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

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:

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

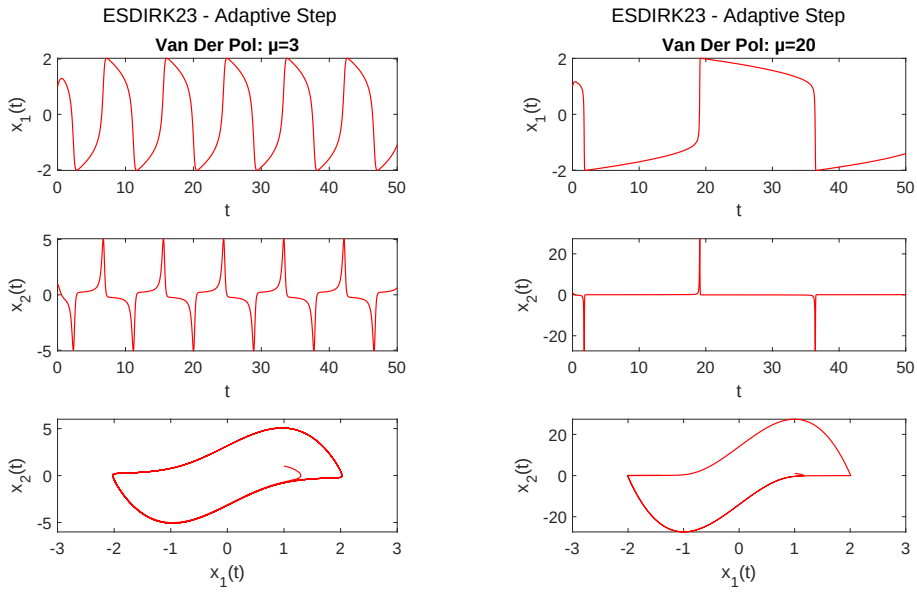
One step of the ESDIRK23 method is summarized in Algorithm 13. The code is listed in Appendix A.1.6, it is derived from the one provided by the teacher [5].

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

Procedure: *ESDIRK23Step*
Data: A, c, d from Butcher tableau, t_n, x_n, h, γ, ϵ, objective function F
Result: $t_{n+1}, x_{n+1}, e_{n+1}$
Compute internal stages
Set $\alpha = 0$ Set $X_1 = x_n$
Set $f_1 = F(t_n, x_n)$
Set $J = \frac{\partial f}{\partial x}(t_n, x_n)$
Set $M = I - h\gamma J$
Set $[L, U] = lu(M)$ `// LU factorize M`
for $i = 2 : 3$ **do**
  $\psi_i = x_n + h\sum_{j=1}^{i-1} a_{ij} f_j$
  $T_i = t_n + c_i h$
  $X_i = x_n + (c_i h) f_1$
  $R_i = X_i - h\gamma f_i - \psi_i$
  while $||R_i|| > \epsilon$ *AND* $\alpha < 1$ **do**
    $\Delta X_i = -U^{-1}(L^{-1}R_i)$ `// Solve using LU-factorization`
    $X_i = X_i + \Delta X_i$
    $f_i = f(T_i, X_i)$
    $R_i = X_i - h\gamma f_i - \psi_i$
    $\alpha = max\{\alpha, \frac{R_{i-1}}{R_i}\}$ `// Monitor convergence rate`
  end
end
Compute next step and estimate error
$e_{n+1} = h\sum_{j=1}^{3} d_j f_j$
$t_{n+1} := T_3$
$x_{n+1} := X_3$

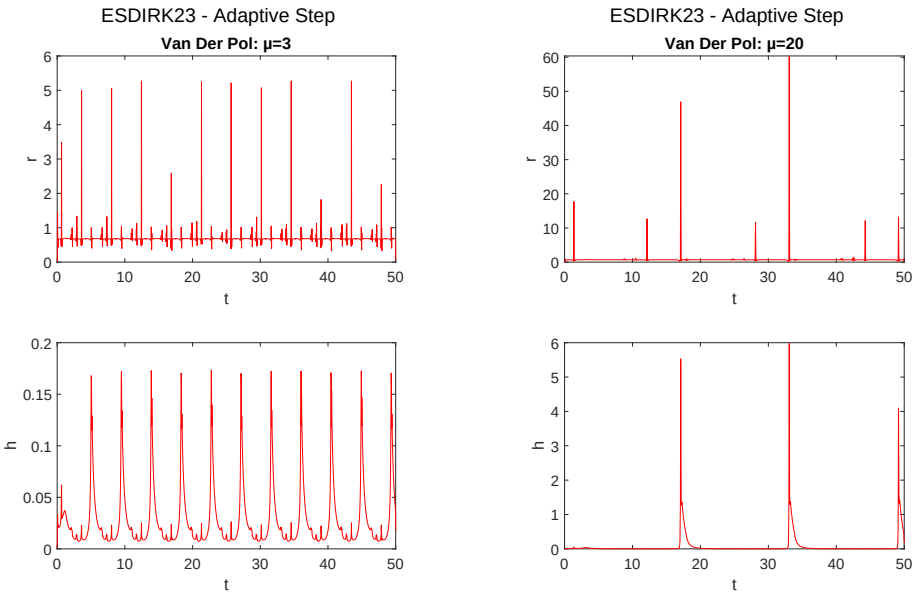
7.1.3 The Van der Pol Problem

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.



((a)) **((b))**

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



((a)) ((b))

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

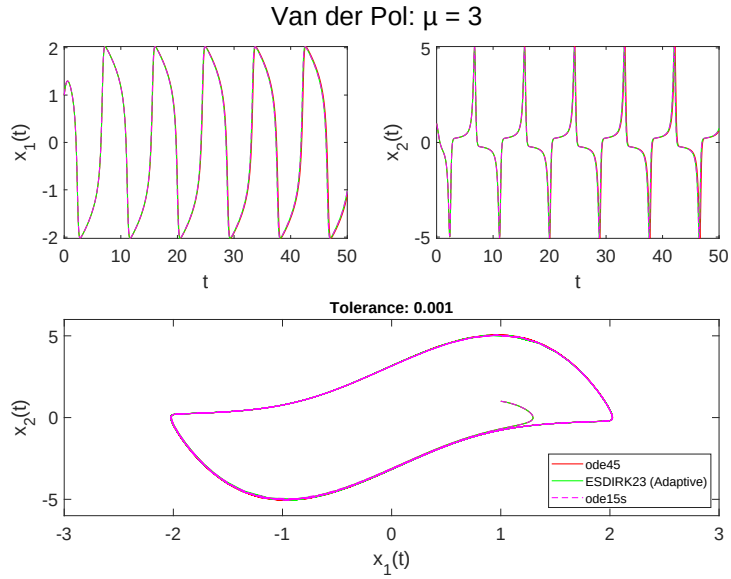
Table 7.4: Comparison of the ESDIRK23 method with Matlab's ODE solvers, *ode45* and *ode15s* for low tolerance of 10^{-7}.

	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

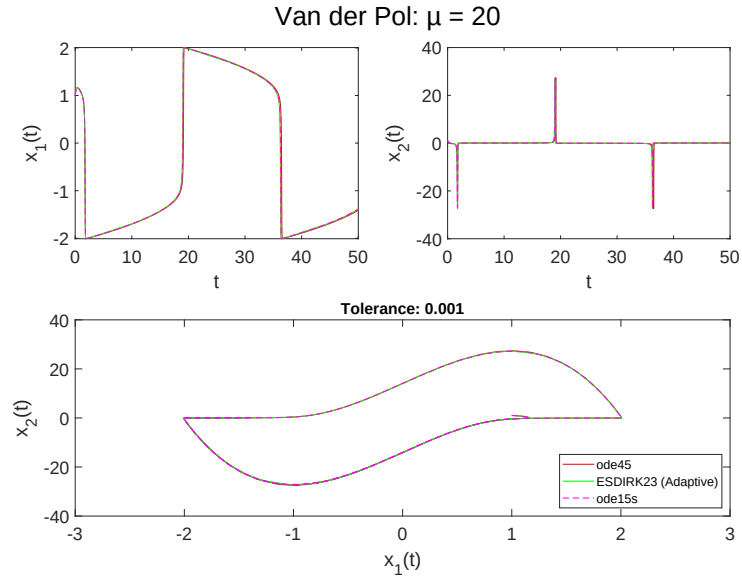
Table 7.5 summarizes the performances of the methods implemented in this exam problem for a low tolerance of 10^{-7}. The DOPRI54 seems to be fastest for both stiff and non-stiff systems, with the classical Runge-Kutta and ESDIRK23 right behind. The Euler methods on the other hand are extremely slow. 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}.

	N. Function		**N. Accepted**		**N. Rejected**	
Method	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$	$\mu = 3$	$\mu = 20$
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

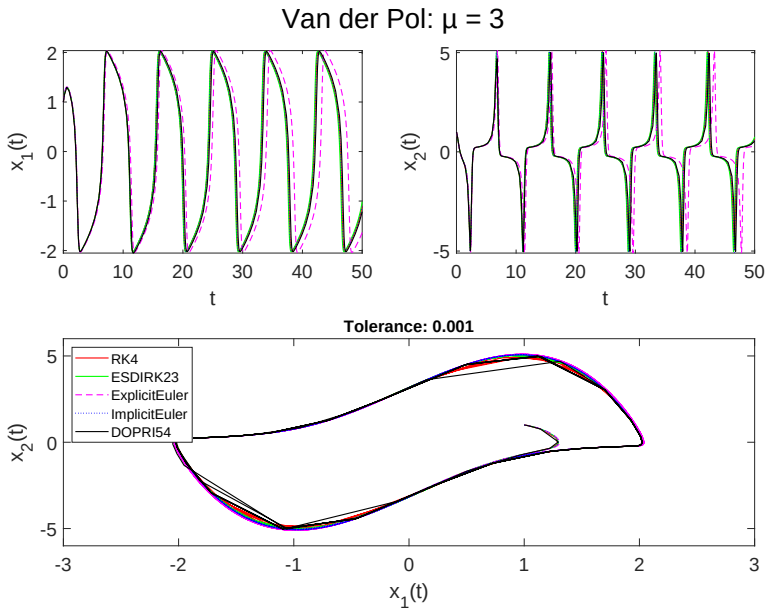


((a))

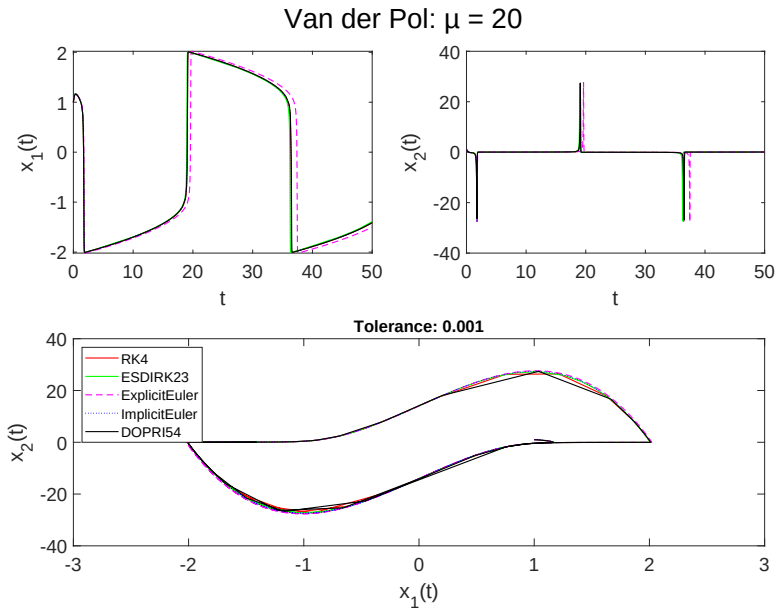


((b))

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



((a))



((b))

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

```
function [T,X] = ExplicitEulerFixedStepSize(fun,t0,tN,N,x0,varargin)
% EXPLICITEULERFIXEDSTEPSIZE Fixed step size Explicit Euler solver for
    systems of ODEs
%
% Syntax:
% [Tout,Xout]=ExplicitEulerFixedStepSize(fun,t0,tN,N,x0,varargin)

% Compute step size and allocate memory
dt = (tN-t0)/N;
nx = size(x0,1);
X = zeros(nx,N+1);
T = zeros(1,N+1);

% Eulers explicit method
T(:,1) = t0;
X(:,1) = x0;
for k=1:N
    f = feval(fun, T(k), X(:,k), varargin{:});
    T(:,k+1) = T(:,k) + dt;
    X(:,k+1) = X(:,k) + f*dt;
end

T = T';
X = X';
```

Listing A.1: Explicit Euler Fixed Step Size

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

```
% [T,X,stat] = ExplicitEulerAdaptiveTimeStep(fun,tspan,x0,h0,abstol,reltol,
    varargin)

% Error controller parameters
epstol = 0.8; % target
facmin = 0.1; % maximum decrease factor
facmax = 5.0; % maximum increase factor

% Integration interval
t0 = tspan(1);
tf = tspan(2);
% Initial conditions
t = t0;
h = h0;
x = x0;

% Output;
T = t;
X = x';

% Store variables
stat.nfun = 0;
stat.nAccept = 0;
stat.nReject = 0;
stat.h = h;
stat.r = [];

% Main algorithm
while t < tf
    if (t+h > tf)
        h = tf - t;
    end
    f = feval(fun,t,x,varargin{:});
    stat.nfun = stat.nfun+1;

    stat.h = [stat.h h];

    AcceptStep = false;
    while  AcceptStep
        % Take step of size h
        x1 = x + h*f;

        % Step Doubling for error estimation
        % Take step of size h/2
        hm = 0.5*h;
        tm = t + hm;
        xm = x + hm*f;
        fm = feval(fun, tm,xm,varargin{:});
        stat.nfun = stat.nfun+1;
        x1hat = xm + hm*fm;

        % Error estimation
        e = x1hat - x1;
        r = max(abs(e)./max(abstol,abs(x1hat).*reltol));

```

```
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;
```

```
    xinit = X(:,k) + f*dt;
    X(:,k+1) = NewtonsMethodODE(funJac,...
    T(:,k), X(:,k), dt, xinit, tol, maxit, varargin{:});
end

% Form a nice table for the result
T = T';
X = X';
```

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

```
function [T,X,stat] = ImplicitEulerAdaptiveStep(funJac,tspan,x0,h0,abstol,
    reltol,varargin)
% ImplicitEulerAdaptiveTimeStep  Implicit Euler solver with adaptive step
% size and error estimation using step doubling for systems of ODEs
%
%   funJac: Function to compute the objective function and it's Jacobian
%   varargin: Parameters for the objective function
%
% Syntax:
% [T,X,stat] = ImplicitEulerAdaptiveStep(funJac,tspan,x0,h0,abstol,reltol,
    varargin)

epstol=0.8; %Target
facmin=0.1; %Maximum decrease factor
facmax=5.0; %Maximum increase factor

%Integration interval
t0=tspan(1);
tf=tspan(2);

%Initial Conditions
t=t0;
h=h0;
x=x0;

tol = 1.0e-8;
maxit = 100;

%Output
T=t;
X=x';

% Store variables
stat.nfun = 0;
stat.nAccept = 0;
stat.nReject = 0;
stat.h = h;
stat.r = [];

%Main algorithm
i=0;
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.

```
function [x,nfun] = NewtonsMethodODE(FunJac, tk, xk, dt, xinit, tol, maxit,
    varargin)
% NewtonsMethodODE Performs Newton's method for approximating next step in
% the implicit Euler method
%
% Syntax: [x,nfun] = NewtonsMethodODE(FunJac,tk,xk,dt,xinit,tol,maxit,
    varargin)

k = 0;
t = tk + dt;
x = xinit;
[f,J] = feval(FunJac,t,x,varargin{:});
nfun = 1; % keep track of function evaluations

R = x - f*dt - xk;
I = eye(length(xk));
while( (k < maxit) && (norm(R,'inf') > tol) )
    k = k+1;
    dRdx = I - J*dt;
    dx = dRdx\R;
    x = x - dx;
    [f,J] = feval(FunJac,t,x,varargin{:});
    nfun = nfun +1;
    R = x - dt*f - xk;
end
```

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

A.1.3 Problem 4 Algorithms

```
function [W,Tw,dW] = StdWienerProcess(T,N,nW,Ns,seed)
% STDWIENERPROCESS Ns realizations of a standard Wiener process
%
% Syntax: [W,Tw,dW] = StdWienerProcess(T,N,Ns,seed)
%
% W     : Standard Wiener process in [0,T]
% Tw    : Time points
% dW    : White noise used to generate the Wiener process
% T     : Final time
% N     : Number of intervals
% nW    : Dimension of W(k)
% Ns    : Number of realizations
% seed  : To set the random number generator (optional)

if nargin == 4
    rng(seed);
end
dt = T/N;
dW = sqrt(dt)*randn(nW,N,Ns);
```

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

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

```
function X=SDEsolverExplicitExplicit(ffun,gfun,T,x0,W,varargin)
% SDEsolverExplicitExplicit Euler-Maruyama solver for systems of SDEs
%
% Syntax: X=SDEsolverExplicitExplicit(ffun,gfun,T,x0,W,varargin)

N = length(T);
nx = length(x0);
X = zeros(nx,N);

X(:,1) = x0;
for k=1:N-1
    f = feval(ffun,T(k),X(:,k),varargin{:});
    g = feval(gfun,T(k),X(:,k),varargin{:});
    dt = T(k+1)-T(k);
    dW = W(:,k+1)-W(:,k);
    psi = X(:,k) + g*dW;
    X(:,k+1) = psi + f*dt;
end
end
```

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

```
function X=SDEsolverImplicitExplicit(ffun,gfun,T,x0,W,varargin)
% SDEsolverImplicitExplicit Implicit-Explicit solver for
% systems of SDEs
%
% Syntax: X=SDEsolverImplicitExplicit(ffun,gfun,T,x0,W,varargin)

tol = 1.0e-8;
maxit = 100;

N = length(T);
nx = length(x0);
X = zeros(nx,N);

X(:,1) = x0;
k=1;
[f, ] = feval(ffun,T(k),X(:,k),varargin{:});

for k=1:N-1
    g = feval(gfun,T(k),X(:,k),varargin{:});
    dt = T(k+1)-T(k);
    dW = W(:,k+1)-W(:,k);
    psi = X(:,k) + g*dW;
    xinit = psi + f*dt;
```

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

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

A.1.4 Problem 5 Algorithms

```
function [Tout,Xout]=ClassicalRungeKuttaSolver(fun,tspan,x0,h,varargin)
% =ClassicalRungeKuttaSolver Fixed step size classical Runge-Kutta
% solver for systems of ODEs
%
% Syntax:
% [Tout,Xout]=ClassicalRungeKuttaSolver(fun,tspan,x0,h,varargin)

% Solver Parameters
s = 4; % Number of stages in ERK method
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
b = [1/6,1/3,1/3,1/6]'; % b-vector in Butcher tableau
c  = [0;1/2;1/2;1]; % c-vector in Butcher tableau

% Parameters related to constant step size
hAT = h*AT;
hb = h*b;
hc = h*c;
% Size parameters
x = x0;
t = tspan(1); % Initial time
tf = tspan(end); % Final time
N = (tf-t)/h; % Number of steps
nx = length(x0); % System size (dim(x))
% Allocate memory
T = zeros(1,s); % Stage T
X = zeros(nx,s); % Stage X
F = zeros(nx,s); % Stage F
Tout = zeros(N+1,1); % Time for output
Xout = zeros(N+1,nx); % States for output

Tout(1) = t;
Xout(1,:) = x';
for n=1:N
    % Stage 1
    T(1) = t;
    X(:,1) = x;
    F(:,1) = feval(fun,T(1),X(:,1),varargin{:});
    % Stage 2,3,...,s
    T(2:s) = t + hc(2:s);
    for i=2:s
```

```
        X(:,i) = x + F(:,1:i-1)*hAT(1:i-1,i);
        F(:,i) = feval(fun,T(i),X(:,i),varargin{:});
    end
    % Next step
    t = t + h;
    x = x + F*hb;
    % Save output
    Tout(n+1) = t;
    Xout(n+1,:) = x';
end

end
```

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

```
function [t1,x1]=ClassicalRungeKuttaStep(fun,t,x,f,h,varargin)

h2=0.5*h;
alpha=h/6;
beta=h/3;

T1=t;
X1=x;
F1=f;

T2=x+h2;
X2=x+h2*F1;
F2=feval(fun,T2,X2,varargin{:});

T3=T2;
X3=x+h2*F2;
F3=feval(fun,T3,X3,varargin{:});

T4=t+h;
X4=x+h*F3;
F4=feval(fun,T4,X4,varargin{:});

t1=T4;

x1=x+alpha*(F1+F4)+beta*(F2+F3);

end
```

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

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

```
% [T,X,stat]=ClassicalRungeKuttaAdaptiveStep(fun,tspan,x0,h0,abstol,reltol,
    varargin)

%Error controller
epstol=0.8;
kpow=0.2;
facmin=0.1;
facmax=5;

%Integration interval
t0=tspan(1);
tf=tspan(2);

%Initial conditions
t=t0;
h=h0;
x=x0;

%Output
T=t;
X=x';

% Storage variables
stat.nfun = 0;
stat.nAccept = 0;
stat.nReject = 0;
stat.h = [];
stat.r = [];

%Algorithm:


i=0;

while t<tf
    i=i+1;

    if (t+h>tf)
        h=tf-t;
    end

    stat.h = [stat.h h];

    f=feval(fun,t,x,varargin{:});
    stat.nfun = stat.nfun +1;

    AcceptStep=false;

    while  AcceptStep
        %take step of size h
        [t1,x1]=ClassicalRungeKuttaStep(fun,t,x,f,h,varargin{:});
        stat.nfun = stat.nfun +3;

        %take step of size h/2
        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-x1;
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,
       varargin)
2  % 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);
```

```
tf = tspan(2);

%Initial conditions
t=t0;
h=h0;
x=x0;

%Output
T=t;
X=x';
E=zeros(max(size(x)),1)';

% Storage variables
stat.nfun = 0;
stat.r = [];
stat.nAccept = 0;
stat.nReject = 0;
stat.h = [];

while t < tf
    if(t+h>tf)
        h = tf-t;
    end
    f = feval(fun,t,x,varargin{:});
    stat.nfun = stat.nfun + 1;

    stat.h = [stat.h h];

    AcceptStep=false;
    while  AcceptStep
        %Take step of size h with built in error estimation
        [t1,x1,e] = DOPRI54Step(fun,t,x,f,h,varargin{:});
        stat.nfun = stat.nfun+6;

        %Error estimation
        r = max(abs(e)./max(abstol,abs(x1).*reltol));

        AcceptStep = (r <= 1.0);
        if AcceptStep
            t = t1;
            x = x1;

            T = [T;t];
            X = [X;x'];
            E = [E;e'];

            stat.nAccept = stat.nAccept+1;
        else
            stat.nReject = stat.nReject+1;
        end
        %Asymptotic step size controller
        h = max(facmin,min((epstol/r)^kpow,facmax))*h;
    end
    stat.r = [stat.r r];
end
```

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

```
function [t1,x1,e] = DOPRI54Step(fun,t,x,f,h,varargin)

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

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

T1 = t;
X1 = x;
F1 = f;

X2 = x+A(2,1)*h*f;
F2 = feval(fun,t+c(2)*h,X2,varargin{:});

X3 = x+h*(A(3,1)*f+A(3,2)*F2);
F3 = feval(fun,t+c(3)*h,X3,varargin{:});

X4 = x+h*(A(4,1)*f + A(4,2)*F2+A(4,3)*F3);
F4 = feval(fun,t+c(4)*h,X4,varargin{:});

X5 = x+h*(A(5,1)*f+A(5,2)*F2+A(5,3)*F3+A(5,4)*F4);
F5 = feval(fun,t+c(5)*h,X5,varargin{:});

X6 = x+h*(A(6,1)*f+A(6,2)*F2+A(6,3)*F3+A(6,4)*F4+A(6,5)*F5);
F6 = feval(fun,t+c(6),X6,varargin{:});

X7 = x+h*(A(7,1)*f+A(7,3)*F3+A(7,4)*F4+A(7,5)*F5+A(7,6)*F6);
F7 = feval(fun,t+c(7),X7,varargin{:});
```

```
51
52 %Output
53 e = h*(d(1)*f+d(3)*F3+d(4)*F4+d(5)*F5+d(6)*F6+d(7)*F7);
54 t1 = t+c(7)*h;
55 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
```

```
rho = 1.0;
cP = 4.186;
k0 = exp(24.6)*60; % in minutes
EaR = 8500; % Ea/R
DeltaH = -560; % Reaction enthalpy
V = 0.105;
CAin = 1.6/2;
CBin = 2.4/2;
Tin = 273.65; % in Kelvin
%Tin = 0.5; % in Celsius

beta = -DeltaH/(rho*cP);

% Get flow rate trajectory
F = feval(flowrate, t);
F = F/1000; % Convert mL to L

% Approximations
CA = CAin + 1/beta*(Tin-T);
CB = CBin + 2/beta*(Tin-T);

% Reaction rate
kT = k0*exp(-EaR*(1/T)); % Ea/(R*T)
r = kT*CA*CB; % (r(CA,CB,T))

RT = beta*r; % Production rate of temperature
RA = -r; % Production rate of A
RB = -2*r; % Production rate of B

% Mass and energy balance considerations of the constant volume reactor:
CAdot = F/V*(CAin-x(1,1)) + RA;
CBdot = F/V*(CBin-x(2,1)) + RB;
Tdot = F/V*(Tin-T) + RT;

xdot = [CAdot, CBdot, Tdot]';

end

function [F] = FlowRate(t)
% FlowRate Computes the flow rate trajectory in mL/min given time t
%
% Syntax:  [F] = FlowRate(t)

if t <= 3.5
    F = 700;
    return;
end
if t <= 5
    F = 600;
    return;
end
if t <= 9
    F = 400;
    return;
end
```

```
if t <= 12
    F = 300;
    return;
end
if t <= 16
    F = 200;
    return;
end
if t <= 18
    F = 300;
    return;
end
if t <= 20
    F = 400;
    return;
end
if t <= 22
    F = 500;
    return;
end
if t <= 24
    F = 600;
    return;
end
if t <= 28
    F = 700;
    return;
end
if t <= 32
    F = 200;
    return;
end
if t <= 35
   F = 700;
   return;
end

F = 0;

end
```

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

A.1.6 Problem 7 Algorithms

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

```
% [Tout,Xout,Gout,info,stats] = ESDIRK23AdaptiveStep(fun,jac,tspan,x0,h0,
    absTol,relTol,varargin)

%ESDIRK23 parameters
gamma = 1-1/sqrt(2);
a31 = (1-gamma)/2;
AT = [0 gamma a31;0 gamma a31;0 0 gamma];
c  = [0; 2*gamma; 1];
b  = AT(:,3);
bhat = [     (6*gamma-1)/(12*gamma); ...
    1/(12*gamma*(1-2*gamma)); ...
    (1-3*gamma)/(3*(1-2*gamma))     ];
d  = b-bhat;
p  = 2;
phat = 3;
s = 3;
t0 = tspan(1);
tf = tspan(2);

% Error and convergence controller
epsilon = 0.8;
tau = 0.1*epsilon;
itermax = 20;
ke0 = 1.0/phat;
ke1 = 1.0/phat;
ke2 = 1.0/phat;
alpharef = 0.3;
alphaJac = -0.2;
alphaLU  = -0.2;
hrmin = 0.01;
hrmax = 10;

info = struct('nStage',    s,        ...
              'absTol',    'dummy',  ...
              'relTol',    'dummy',  ...
              'iterMax',   itermax,  ...
              'tspan',     tspan,    ...
              'nFun',      0,  ...
              'nJac',      0,  ...
              'nLU',       0,  ...
              'nBack',     0,  ...
              'nStep',     0,  ...
              'nAccept',   0,  ...
              'nFail',     0,  ...
              'nDiverge',  0,  ...
              'nSlowConv', 0);

%% Main integrator
nx = size(x0,1);
F = zeros(nx,s);
t = t0;
x = x0;
h = h0;

[F(:,1),g]  = feval(fun,t,x,varargin{:});
```

```
60 info.nFun = info.nFun+1;
61 [dfdx,dgdx] = feval(jac,t,x,varargin{:});
62 info.nJac = info.nJac+1;
63 FreshJacobian = true;
64 if (t+h)>tf
65     h = tf-t;
66 end
67 hgamma = h*gamma;
68 dRdx = dgdx - hgamma*dfdx;
69 [L,U,pivot] = lu(dRdx,'vector');
70 info.nLU = info.nLU+1;
71 hLU = h;
72
73 FirstStep = true;
74 ConvergenceRestriction = false;
75 PreviousReject = false;
76 iter = zeros(1,s);
77
78 % Initialize output
79 Nsize = 100;
80 Tout = zeros(Nsize,1);
81 Xout = zeros(Nsize,nx);
82 Gout = zeros(Nsize,nx);
83
84 Tout(1,1) = t;
85 Xout(1,:) = x.';
86 Gout(1,:) = g.';
87
88 while t<tf
89     info.nStep = info.nStep+1;
90
91     % ESDIRK23
92     i=1;
93     diverging = false;
94     SlowConvergence = false; % carsten
95     alpha = 0.0;
96     Converged = true;
97     while (i<s) && Converged
98         % Stage i=2,...,s of the ESDIRK Method
99         i=i+1;
100        phi = g + F(:,1:i-1)*(h*AT(1:i-1,i));
101
102        % Initial guess for the state
103         if i==2
104             dt = c(i)*h;
105             G = g + dt*F(:,1);
106             X = x + dgdx\(G-g);
107         else
108             dt = c(i)*h;
109             G  = g + dt*F(:,1);
110             X  = x + dgdx\(G-g);
111         end
112        T = t+dt;
113
114        [F(:,i),G] = feval(fun,T,X,varargin{:});
```

```
115         info.nFun = info.nFun+1;
116         R = G - hgamma*F(:,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 - hgamma*F(:,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
            iterations)
154     stats.SlowConv(nstep)  = SlowConvergence*i;
155     iter(:) = 0;
156
157     % Error and Convergence Controller
158     if Converged
159         % Error estimation
160         e = F*(h*d);
161         r = norm(e./(absTol + abs(G).*relTol),inf);
162         AcceptCurrentStep = (r<=1.0);
163         r = max(r,eps);
164         stats.r(nstep) = r;
165         % Step Length Controller
166         if AcceptCurrentStep
167             stats.AcceptStep(nstep) = true;
168             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             hgamma = 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         hgamma = h*gamma;
260         dRdx = dgdx - hgamma*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
```

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

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

A.2 Drivers

A.2.1 Problem 1 Driver

```
%% Problem 1
close all; clear all; clc;
%1. Provide the analytical solution to the test equation

lambda = -1;
x0=1;
t = linspace(0,20,100);

% Plot the analytical solution
plot(t, exp(x0.*lambda.*t));

%% Exercise 1.3: Local and global truncation errors for the test eq.
%% Exercise 1.3a) Explicit Euler method (fixed step size)
close all; clear all; clc

lambda = -1;
x0=1;

t0=0;
tN=10;
N=50;
h = (tN-t0)/N;

[T,X] = ExplicitEulerFixedStepSize(@TestEq,t0,tN,N,x0,lambda);

% Show analytical solution vs the approximations
plot(T, exp(x0.*lambda.*T)), hold on,
plot(T,X);
xlabel('t')
ylabel('x(t)')
title('Test Equation')
legend('Analytical Solution','Explicit Euler Fixed Step Size')

% Error estimation

% Global error:
e_glob = X-exp(x0.*lambda.*T);
```

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

```
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
      in Butcher tableau
135 solver.b = [1/6,1/3,1/3,1/6]'; % b-vector in Butcher tableau
136 solver.c  = [0;1/2;1/2;1]; % c-vector in Butcher tableau
137
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,
            lambda);
147     e_loc_runge(i)= abs(X_runge(end)-X_runge(end-1)*exp(H(i)*lambda));
148
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
152     [T,X_IE] = ImplicitEulerFixedStepSize2D(@TestEqJac,0,H(i),1,x0,lambda);
153     e_loc_IE(i)= abs(X_IE(end)-  exp(lambda*H(i))*X_IE(end-1));
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\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
195 %% Exercise 1.5. Plot the global error at t=1.0 for the three methods.
196 clear all
197
198 lambda = -1;
199 x0=1;
```

```
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,
          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
248 ylabel('|e|', 'FontSize', 24)
249 xlabel('h', 'FontSize', 24)
250 title('Global Error', 'FontSize', 28)
251 legend('Explicit Euler','Implicit Euler','Classical Runge Kutta',...
252     'Location', 'NorthWest','fontsize',16)
253 set(gca,'fontsize',18)
```

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

```
max_IE=max(max(IE_Astable));

runge_Astable = absR_runge(:,ind_ltzero);
max_runge = max(max(runge_Astable));


fprintf(...
'The highest value for Re(z)<0 in the stability region for Explicit Euler is: %f and thus it is not A-stable\n',max_EE)
fprintf(...
'The highest value for Re(z)<0 in the stability region for Implicit Euler is: %f and thus it is A-stable\n',max_IE)
fprintf(...
'The highest value for Re(z)<0 in the stability region for Classical Runge Kutta is: %f and thus it is not A-stable\n\n',max_runge)


%Which of the methods are L-stable?

%Lets check some low values
z= -10000-i*10000;

R_EE = 1+z ;
R_IE = 1/(1-z);
absR_EE = abs(R_EE);
absR_IE = abs(R_IE);


tmp = (I-z*A)\e;
R_runge = 1 + z*b'*tmp;
absR_runge = abs(R_runge);

fprintf('The stability region for Explicit Euler when z->inf is: %f and thus it is not L-stable\n',absR_EE)
fprintf('The stability region for Implicit Euler when z->inf is: %f and thus it is L-stable\n',absR_IE)
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

```
%% Problem 2: Explicit ODE Solver

% ------------------------------------------------------------
%% Exercise 2.2-4: Explicit Euler Fixed Step Size
close all; clear all; clc;


% Van der Pol problem parameters
```

```
x0 = [1.0,1.0]';
mu = [3, 20];
N = 10000; % Change this to increase/decrease step size
tspan = [0,50];



for N = [3000,10000]

    [T,X] = ExplicitEulerFixedStepSize(@VanderPolfunjac,tspan(1),tspan(2),N,
        x0,mu(1));
    [T2,X2] = ExplicitEulerFixedStepSize(@VanderPolfunjac,tspan(1),tspan(2),
        N,x0,mu(2));

    % Plot results in two figures for each mu
    fs = 21; % fontsize for labels
    figure; f = gcf;
    ax=subplot(3,1,1);
    plot(T,X(:,1),'r','LineWidth',1.5);
    xlabel('t','FontSize',fs),ylabel('x_1(t)','FontSize',fs),
    ylim([-3 3]),xlim([0 50]), set(gca,'fontsize',fs-3)
    subtitle('Van der Pol: μ=3','fontsize',fs-2,'fontweight','bold')
    %title('Explicit Euler Fixed Step Size')
    ax2=subplot(3,1,2);
    plot(T,X(:,2),'r','LineWidth',1.5);
    xlabel('t','FontSize',fs),ylabel('x_2(t)','FontSize',fs),
    ylim([-6 6]),xlim([0 50]), set(gca,'fontsize',fs-3)
    ax3=subplot(3,1,3);
    plot(X(:,1),X(:,2),'r','LineWidth',1.5),
    ylabel('x_2(t)','FontSize',fs),xlabel('x_1(t)','FontSize',fs),ylim([-8
        8])
    %set(gcf,'Position',[100 100 550 550])
    sgtitle({'Explicit Euler - Fixed Step'},'FontSize', fs+4)

    figure; f2 = gcf;
    ax4=subplot(3,1,1);
    plot(T2,X2(:,1),'r','LineWidth',1.5);
    xlabel('t','FontSize',fs),ylabel('x_1(t)','FontSize',fs),
    ylim([-3 3]),xlim([0 50]), set(gca,'fontsize',fs-3)
    subtitle('Van der Pol: μ=20','fontsize',fs-2,'fontweight','bold')
    ax5=subplot(3,1,2);
    plot(T2,X2(:,2),'r','LineWidth',1.5),
    xlabel('t','FontSize',fs),ylabel('x_2(t)','FontSize',fs),
    ylim([-30 30]),xlim([0 50]), set(gca,'fontsize',fs-3)
    ax6=subplot(3,1,3);
    plot(X2(:,1),X2(:,2),'r','LineWidth',1.5),ylim([-33 33]),
    ylabel('x_2(t)','FontSize',fs),xlabel('x_1(t)','FontSize',fs)
    sgtitle({'Explicit Euler - Fixed Step'},'FontSize', fs+4)
    set(gca,'fontsize',fs-3)
    set([ax,ax2,ax3,ax4,ax5,ax6],'fontsize',fs-3)

end


% -------------------------------------------------------------------
```

```
%% Exercise 2.3-4 Explicit Euler Adaptive Time Step and Error Estimation
close all; clear all; clc;

x0 = [1.0,1.0]';
mu = [3, 20];
abstol = 1e-5;
reltol = 1e-5;
t0 = 0;
fs = 21;
tspan = [0,50];

h0 = 0.001; % h0 = 10^-2;

[T,X,stat] = ExplicitEulerAdaptiveTimeStep(@VanderPolfunjac,...
    tspan,x0,h0,abstol,reltol,mu(1));
[T2,X2,stat2] = ExplicitEulerAdaptiveTimeStep(@VanderPolfunjac,...
    tspan,x0,h0,abstol,reltol,mu(2));


% Plot results in two figures for each mu
figure; f3 = gcf;
ax = subplot(3,1,1);
plot(T,X(:,1),'r','LineWidth',1.5);
xlabel('t','FontSize',fs), ylabel('x_1(t)','FontSize',fs),
ylim([-3 3]),xlim([0 50])
subtitle({'Van Der Pol: \mu=3'},'fontsize',fs-2,'fontweight','bold')
%set(gca,'fontsize',fs-3)
ax2 = subplot(3,1,2);
plot(T,X(:,2),'r','LineWidth',1.5);
xlabel('t','FontSize',fs), ylabel('x_2(t)','FontSize',fs),
ylim([-6 6]),xlim([0 50])
ax3 = subplot(3,1,3);
plot(X(:,1),X(:,2),'r','LineWidth',1.5),ylim([-8 8]),
ylabel('x_2(t)','FontSize',fs),xlabel('x_1(t)','FontSize',fs)
sgtitle({'Explicit Euler - Adaptive Step'}, 'FontSize', fs+4)


figure;f4=gcf;
ax4=subplot(3,1,1);
line3 = plot(T2,X2(:,1),'r','LineWidth',1.5); ylabel('x_1(t)','FontSize',fs)
    ,
xlabel('t','FontSize',fs),ylim([-3 3]),xlim([0 50])
subtitle({'Van Der Pol: \mu=20'},'fontsize',fs-2,'fontweight','bold')
ax5 =subplot(3,1,2);
plot(T2,X2(:,2),'r','LineWidth',1.5),ylabel('x_2(t)','FontSize',fs),
xlabel('t','FontSize',fs),ylim([-30 30]),xlim([0 50])
ax6 = subplot(3,1,3);
plot(X2(:,1),X2(:,2),'r','LineWidth',1.5),ylim([-30 30]),
ylabel('x_2(t)','FontSize',fs),xlabel('x_1(t)','FontSize',fs)
sgtitle({'Explicit Euler - Adaptive Step'},'FontSize',fs+4)
set([ax,ax2,ax3,ax4,ax5,ax6],'fontsize',fs-3)


% Look at error estimation and step sizes
figure;f5=gcf;
```

```
ax=subplot(2,1,1);
plot(linspace(0,50, length(stat.r)), stat.r, 'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('r','fontsize',fs), %ylim([-0.1 1])
subtitle({'Van Der Pol: μ=3'},'fontsize',fs-3,'fontweight','bold')
ax2=subplot(2,1,2);
plot(linspace(0,50, length(stat.h)), stat.h, 'r','LineWidth', 1),
xlabel('t'),ylabel('h'), %ylim([-0.01 0.04])
sgtitle('Explicit Euler - Adaptive Step','fontsize',fs+4)

figure; f6=gcf;
ax3=subplot(2,1,1);
plot(linspace(0,50, length(stat2.r)), stat2.r,'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('r','fontsize',fs-2), %ylim([0 1])
subtitle({'Van Der Pol: μ=20'},'fontsize',fs-3,'fontweight','bold')
ax4=subplot(2,1,2);
plot(linspace(0,50, length(stat2.h)), stat2.h, 'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('h','fontsize',fs), %ylim([-0.01 0.3])%
sgtitle('Explicit Euler - Adaptive Step','fontsize',fs+4)
set([ax,ax2,ax3,ax4],'fontsize',fs-3)


fprintf('Van der Pol Explicit Euler Adaptive Step Size μ=3\n')
T = table(reltol, stat.nfun, stat.nAccept, stat.nReject, 'VariableNames',
    ...
    {'Tolerance','N. Func','N. Accept','N. Reject'});
disp(T)
fprintf('Van der Pol Explicit Euler Adaptive Step Size μ=20\n')
T = table(reltol, stat2.nfun, stat2.nAccept, stat2.nReject, 'VariableNames',
    ...
    {'Tolerance','N. Func','N. Accept','N. Reject'});
disp(T)


%% Exercise 2.5  Compare with Matlabs ODE solvers using different tolerances
    and step sizes
close all; clear all; clc;

fs = 22;
plot_results = true; % Change to not plot figures
mu = [3,20];
x0 = [1.0;1.0];
h0 = 0.001;
tspan = [0 50];

abstol = [1e-3, 1e-5, 1e-7];
reltol = abstol;

n = length(abstol);
% Initialize stat variables
acc3 = zeros(n,1); acc20 = zeros(n,1);
rej3 = zeros(n,1); rej20 = zeros(n,1);
nfun3 = zeros(n,1); nfun20 = zeros(n,1);


N = 5000;
```

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

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

```
T=table(reltol',statode45_2(3,:)', statode45_2(1,:)',statode45_2(2,:)', '
    VariableNames', varnames);
disp(T)
fprintf('Van der Pol ODE15s μ=3\n')
T=table(reltol',statode15s(3,:)',statode15s_2(1,:)',statode15s_2(2,:)', '
    VariableNames', varnames);
disp(T)
fprintf('Van der Pol ODE15s μ=20\n')
T=table(reltol',statode15s_2(3,:)',statode15s(1,:)',statode15s(2,:)','
    VariableNames', varnames);
disp(T)
```

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

A.2.3 Problem 3 Driver

```
%% Problem 3: Implicit ODE Solver

% ------------------------------------------------------------
%% Exercise 3.3-4: Explicit Euler Fixed Step Size
close all; clear all; clc;


% Van der Pol problem parameters
x0 = [1.0,1.0]';
mu = [3, 20];
tspan = [0,50];

for N=[3000,10000] % Steps less than 10000 gives a large error for mu=20
    [T,X] = ImplicitEulerFixedStepSize(@VanderPolfunjac,tspan(1),tspan(2),N,
        x0,mu(1));
    [T2,X2] = ImplicitEulerFixedStepSize(@VanderPolfunjac,tspan(1),tspan(2),
        N,x0,mu(2));


    % Plot results in two figures for each mu
    fs = 20; % fontsize for labels
    figure; f = gcf;
    ax=subplot(3,1,1);
    plot(T,X(:,1),'r','LineWidth',1.5);
    xlabel('t','FontSize',fs),ylabel('x_1(t)','FontSize',fs),
    ylim([-3 3]),xlim([0 50]), set(gca,'fontsize',fs-3)
    subtitle('Van der Pol: μ=3','fontsize',fs-2,'fontweight','bold')
    ax2=subplot(3,1,2);
    plot(T,X(:,2),'r','LineWidth',1.5);
    xlabel('t','FontSize',fs),ylabel('x_2(t)','FontSize',fs),
    ylim([-6 6]),xlim([0 50]), set(gca,'fontsize',fs-3)
    ax3=subplot(3,1,3);
    plot(X(:,1),X(:,2),'r','LineWidth',1.5),
    xlim([-3 3])
```

```
33      ylabel('x_2(t)','FontSize',fs),xlabel('x_1(t)','FontSize',fs),ylim([-8
            8])
34      sgtitle({'Implicit Euler - Fixed Step'},'FontSize', fs+4)
35
36      figure; f2 = gcf;
37      ax4=subplot(3,1,1);
38      plot(T2,X2(:,1),'r','LineWidth',1.5);
39      xlabel('t','FontSize',fs),ylabel('x_1(t)','FontSize',fs),
40      ylim([-3 3]),xlim([0 50]), set(gca,'fontsize',fs-3)
41      subtitle('Van der Pol: μ=20','fontsize',fs-2,'fontweight','bold')
42      ax5=subplot(3,1,2);
43      plot(T2,X2(:,2),'r','LineWidth',1.5),
44      xlabel('t','FontSize',fs),ylabel('x_2(t)','FontSize',fs),
45      ylim([-30 30]),xlim([0 50]), set(gca,'fontsize',fs-3)
46      ax6=subplot(3,1,3);
47      plot(X2(:,1),X2(:,2),'r','LineWidth',1.5),ylim([-33 33]),
48      xlim([-3 3])
49      ylabel('x_2(t)','FontSize',fs),xlabel('x_1(t)','FontSize',fs)
50      sgtitle({'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: μ=3'},'fontsize',fs-2,'fontweight','bold')
86  ax2 = subplot(3,1,2);
```

```
plot(T,X(:,2),'r','LineWidth',1.5);
xlabel('t','FontSize',fs), ylabel('x_2(t)','FontSize',fs),
%ylim([-6 6]),xlim([0 50])
ax3 = subplot(3,1,3);
plot(X(:,1),X(:,2),'r','LineWidth',1.5),ylim([-8 8]),
ylabel('x_2(t)','FontSize',fs),xlabel('x_1(t)','FontSize',fs)
sgtitle({'Implicit Euler - Adaptive Step'}, 'FontSize', fs+4)


figure;f4=gcf;
ax4=subplot(3,1,1);
line3 = plot(T2,X2(:,1),'r','LineWidth',1.5); ylabel('x_1(t)','FontSize',fs)
    ,
xlabel('t','FontSize',fs),ylim([-3 3]),xlim([0 50])
subtitle({'Van Der Pol: μ=20'},'fontsize',fs-2,'fontweight','bold')
ax5 =subplot(3,1,2);
plot(T2,X2(:,2),'r','LineWidth',1.5),ylabel('x_2(t)','FontSize',fs),
xlabel('t','FontSize',fs),ylim([-30 30]),xlim([0 50])
ax6 = subplot(3,1,3);
plot(X2(:,1),X2(:,2),'r','LineWidth',1.5),ylim([-30 30]),
ylabel('x_2(t)','FontSize',fs),xlabel('x_1(t)','FontSize',fs)
sgtitle({'Implicit Euler - Adaptive Step'},'FontSize',fs+4)


% Look at error estimation and step sizes
figure;f5=gcf;
ax=subplot(2,1,1);
plot(linspace(0,50, length(stat.r)), stat.r, 'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('r','fontsize',fs), %ylim([-0.1 1])
subtitle({'Van Der Pol: μ=3'},'fontsize',fs-3,'fontweight','bold')
ax2=subplot(2,1,2);
plot(linspace(0,50, length(stat.h)), stat.h, 'r','LineWidth', 1),
xlabel('t','FontSize',fs),ylabel('h','FontSize',fs), ylim([0 0.04])
sgtitle('Implicit Euler - Adaptive Step','fontsize',fs+4)

figure; f6=gcf;
ax3=subplot(2,1,1);
plot(linspace(0,50, length(stat2.r)), stat2.r,'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('r','fontsize',fs-2), %ylim([0 1])
subtitle({'Van Der Pol: μ=20'},'fontsize',fs-3,'fontweight','bold')
ax4=subplot(2,1,2);
plot(linspace(0,50, length(stat2.h)), stat2.h, 'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('h','fontsize',fs), %ylim([-0.01 0.3])%
sgtitle('Implicit Euler - Adaptive Step','fontsize',fs+4)

set([ax,ax2,ax3,ax4],'fontsize',fs-3)

fprintf('Van der Pol Explicit Euler Adaptive Step Size μ=3\n')
T = table(reltol, stat.nfun, stat.nAccept, stat.nReject, 'VariableNames',
    ...
    {'Tolerance','N. Func','N. Accept','N. Reject'});
disp(T)
fprintf('Van der Pol Explicit Euler Adaptive Step Size μ=20\n')
T = table(reltol, stat2.nfun, stat2.nAccept, stat2.nReject, 'VariableNames',
     ...
```

```
    {'Tolerance','N. Func','N. Accept','N. Reject'});
disp(T)


%% Exercise 3.5  Compare using different tolerances and step sizes
close all; clear all; clc;

fs = 20;
plot_results = true;
mu = [3,20];
x0 = [1.0;1.0];
h0 = 0.001;
tspan = [0 50];

abstol = [1e-3, 1e-5, 1e-7];
reltol = abstol;

n = length(abstol);
% Initialize stat variables
acc3 = zeros(n,1); acc20 = zeros(n,1);
rej3 = zeros(n,1); rej20 = zeros(n,1);
nfun3 = zeros(n,1); nfun20 = zeros(n,1);

N = 5000;


% Add fixed euler to compare
[Tf,Xf] = ImplicitEulerFixedStepSize(@VanderPolfunjac,tspan(1),tspan(2),N,x0
    ,mu(1));
[T2f,X2f] = ImplicitEulerFixedStepSize(@VanderPolfunjac,tspan(1),tspan(2),N,
    x0,mu(2));

for i = 1:n

    % Matlab's ode solvers
    options = odeset("RelTol",reltol(i),"AbsTol",abstol(i));%,'Stats','on');
    [Tmu3,Xmu3, statODE45]=ode45(@VanderPolfunjac,tspan,x0,options,mu(1));
    [Tmu20,Xmu20, statODE45_2]=ode45(@VanderPolfunjac,tspan,x0,options,mu(2)
        );
    [T2mu20,X2mu20,statODE15s]=ode15s(@VanderPolfunjac,tspan,x0,options,mu
        (2));
    [T2mu3,X2mu3,statODE15s_2]=ode15s(@VanderPolfunjac,tspan,x0,options,mu
        (1));

    % Using same initial step size as above
    [T,X,stat] = ImplicitEulerAdaptiveStep(...
        @VanderPolfunjac,tspan,x0,h0,abstol(i),reltol(i),mu(1));
    [T2,X2,stat2] = ImplicitEulerAdaptiveStep(...
        @VanderPolfunjac,tspan,x0,h0,abstol(i),reltol(i),mu(2));

    if plot_results

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

```
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 μ=3\n')
250 T=table(reltol', nfun3, acc3, rej3, 'VariableNames', varnames);
251 disp(T)
252 fprintf('Van der Pol Implicit Euler Adaptive Step Size μ=20\n')
253 T=table(reltol', nfun20, acc20, rej20, 'VariableNames', varnames);
254 disp(T)
255 fprintf('Van der Pol ODE45 μ=3\n')
256 T=table(reltol',statode45(3,:)',statode45(1,:)',statode45(2,:)', '
        VariableNames', varnames);
257 disp(T)
258 fprintf('Van der Pol ODE45 μ=20\n')
259 T=table(reltol',statode45_2(3,:)', statode45_2(1,:)',statode45_2(2,:)', '
        VariableNames', varnames);
260 disp(T)
261 fprintf('Van der Pol ODE15s μ=3\n')
262 T=table(reltol',statode15s(3,:)',statode15s_2(1,:)',statode15s_2(2,:)', '
        VariableNames', varnames);
263 disp(T)
264 fprintf('Van der Pol ODE15s μ=20\n')
265 T=table(reltol',statode15s_2(3,:)',statode15s(1,:)',statode15s(2,:)','
        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;
```

```
mu = [3,20];
sigma = 0.5;
x0 = [0.5; 0.5];

for j=1:2

    %tf = 5*mu(j);
    tf = 5*mu(j);

    k = 1;
    p = [mu(j); sigma];
    for nsteps = N
        % Get standard wiener process parameter (random variable)
        [W,T, ]=StdWienerProcess(tf,nsteps,nw,Ns,seed);


        % Implicit-Explicit method: STATE INDEPENDENT DIFFUSION

        X = zeros(length(x0),nsteps+1,Ns);
        for i=1:Ns
            X(:,:,i) = SDEsolverImplicitExplicit(...
            @VanderPolDrift,@VanderPolDiffusion1,...
            T,x0,W(:,:,i),p);
        end
        Xd = SDEsolverImplicitExplicit(...
            @VanderPolDrift,@VanderPolDiffusion1,...
            T,x0,W(:,:,i),[mu(j); 0.0]); % without sigma - to show only the
                drift term


        % Implicit-Explicit method: STATE DEPENDENT DIFFUSION
        for i=1:Ns
            X2(:,:,i) = SDEsolverImplicitExplicit(...
            @VanderPolDrift,@VanderPolDiffusion2,...
            T,x0,W(:,:,i),p);
        end
        Xd2 = SDEsolverImplicitExplicit(...
            @VanderPolDrift,@VanderPolDiffusion2,...
            T,x0,W(:,:,i),[mu(j); 0.0]);

        % Plot results
        title = {'Implicit-Explicit SDE: Van der Pol','State Independent
            Diffusion'};
        title2 = {'Implicit-Explicit SDE: Van der Pol','State Dependent
            Diffusion'};

        fig(k,j) = plotVanDerPolResults(T,X,Xd,title);
        fig2(k,j) = plotVanDerPolResults(T,X2,Xd2,title2);
        k = k+1;
        clear X X2 Xd Xd2
    end
end
```

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

A.2.5 Problem 5 Driver

```
%% Problem 5: Classical Runge Kutta

%% Exercise 5.2-5.4 Classical Runge-Kutta method with fixed step size
% on the Van der Pol problem
close all; clear all;clc

%Test your algorithms on the van der pol problem (mu=3, mu=20, x0=[1,1]')

mu = [3,20];
tspan = [0,50];
x0 = [1,1]';


for N = [3000,10000]
    h0 = tspan(2)/N;

    [T,X]=ClassicalRungeKuttaSolver(@VanderPolfunjac,tspan,x0,h0,mu(1));

    [T2,X2]=ClassicalRungeKuttaSolver(@VanderPolfunjac,tspan,x0,h0,mu(2));

    % Plot results in two figures for each mu
    fs = 22; % fontsize for labels
    figure; f = gcf;
    ax=subplot(3,1,1);
    plot(T,X(:,1),'r','LineWidth',1.5);
    xlabel('t','FontSize',fs),ylabel('x_1(t)','FontSize',fs),
    ylim([-3 3]),xlim([0 50]), set(gca,'fontsize',fs-3)
    subtitle('Van der Pol: \mu=3','fontsize',fs-2,'fontweight','bold')
    %title('Explicit Euler Fixed Step Size')
    ax2=subplot(3,1,2);
    plot(T,X(:,2),'r','LineWidth',1.5);
    xlabel('t','FontSize',fs),ylabel('x_2(t)','FontSize',fs),
    ylim([-6 6]),xlim([0 50]), set(gca,'fontsize',fs-3)
    ax3=subplot(3,1,3);
    plot(X(:,1),X(:,2),'r','LineWidth',1.5),
    ylabel('x_2(t)','FontSize',fs),xlabel('x_1(t)','FontSize',fs),ylim([-8
        8])
    sgtitle({'Classical Runge-Kutta - Fixed Step'},'FontSize', fs+4)

    figure; f2 = gcf;
    ax4=subplot(3,1,1);
    plot(T2,X2(:,1),'r','LineWidth',1.5);
    xlabel('t','FontSize',fs),ylabel('x_1(t)','FontSize',fs),
    ylim([-3 3]),xlim([0 50]), set(gca,'fontsize',fs-3)
    subtitle('Van der Pol: \mu=20','fontsize',fs-2,'fontweight','bold')
    ax5=subplot(3,1,2);
    plot(T2,X2(:,2),'r','LineWidth',1.5),
    xlabel('t','FontSize',fs),ylabel('x_2(t)','FontSize',fs),
    ylim([-30 30]),xlim([0 50]), set(gca,'fontsize',fs-3)
    ax6=subplot(3,1,3);
    plot(X2(:,1),X2(:,2),'r','LineWidth',1.5),ylim([-33 33]),
```

```
    ylabel('x_2(t)','FontSize',fs),xlabel('x_1(t)','FontSize',fs)
    sgtitle({'Classical Runge-Kutta - Fixed Step'},'FontSize', fs+4)
    set(gca,'fontsize',fs-3)

end


%% Exercise 5.3-4 Classical RK Adaptive Time Step and Error Estimation
close all; clear all; clc;

x0 = [1.0,1.0]';
mu = [3, 20];
abstol = 1e-5;
reltol = 1e-5;
t0 = 0;
fs = 22;
tspan = [0,50];

h0 = 0.001; % h0 = 10^-2;

[T,X,stat]=ClassicalRungeKuttaAdaptiveStep(@VanderPolfunjac,tspan,x0,h0,
    abstol,reltol,mu(1));
[T2,X2,stat2]=ClassicalRungeKuttaAdaptiveStep(@VanderPolfunjac,tspan,x0,h0,
    abstol,reltol,mu(2));


% Plot results in two figures for each mu
figure; f3 = gcf;
ax = subplot(3,1,1);
plot(T,X(:,1),'r','LineWidth',1.5);
xlabel('t','FontSize',fs), ylabel('x_1(t)','FontSize',fs),
ylim([-3 3]),xlim([0 50])
subtitle({'Van Der Pol: μ=3'},'fontsize',fs-2,'fontweight','bold')

ax2 = subplot(3,1,2);
plot(T,X(:,2),'r','LineWidth',1.5);
xlabel('t','FontSize',fs), ylabel('x_2(t)','FontSize',fs),
ylim([-6 6]),xlim([0 50])
ax3 = subplot(3,1,3);
plot(X(:,1),X(:,2),'r','LineWidth',1.5),ylim([-8 8]),
ylabel('x_2(t)','FontSize',fs),xlabel('x_1(t)','FontSize',fs)
sgtitle({'Classical Runge-Kutta - Adaptive Step'}, 'FontSize', fs+4)


figure;f4=gcf;
ax4=subplot(3,1,1);
line3 = plot(T2,X2(:,1),'r','LineWidth',1.5); ylabel('x_1(t)','FontSize',fs)
    ,
xlabel('t','FontSize',fs),ylim([-3 3]),xlim([0 50])
subtitle({'Van Der Pol: μ=20'},'fontsize',fs-2,'fontweight','bold')
ax5 =subplot(3,1,2);
plot(T2,X2(:,2),'r','LineWidth',1.5),ylabel('x_2(t)','FontSize',fs),
xlabel('t','FontSize',fs),ylim([-30 30]),xlim([0 50])
ax6 = subplot(3,1,3);
plot(X2(:,1),X2(:,2),'r','LineWidth',1.5),ylim([-30 30]),
```

```
ylabel('x_2(t)','FontSize',fs),xlabel('x_1(t)','FontSize',fs)
sgtitle({'Classical Runge-Kutta - Adaptive Step'},'FontSize',fs+4)
set([ax,ax2,ax3,ax4,ax5,ax6],'fontsize',fs-3)


% Look at error estimation and step sizes
figure;f5=gcf;
ax=subplot(2,1,1);
plot(linspace(0,50, length(stat.r)), stat.r, 'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('r','fontsize',fs), ylim([0 1.1])
subtitle({'Van Der Pol: μ=3'},'fontsize',fs-3,'fontweight','bold')
ax2=subplot(2,1,2);
plot(linspace(0,50, length(stat.h)), stat.h, 'r','LineWidth', 1),
xlabel('t'),ylabel('h'), %ylim([-0.01 0.04])
sgtitle('Classical Runge-Kutta - Adaptive Step','fontsize',fs+4)

figure; f6=gcf;
ax3=subplot(2,1,1);
plot(linspace(0,50, length(stat2.r)), stat2.r,'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('r','fontsize',fs-2), ylim([0 1.1])
subtitle({'Van Der Pol: μ=20'},'fontsize',fs-3,'fontweight','bold')
ax4=subplot(2,1,2);
plot(linspace(0,50, length(stat2.h)), stat2.h, 'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('h','fontsize',fs), ylim([0 0.2])%
sgtitle('Classical Runge-Kutta - Adaptive Step','fontsize',fs+4)

set([ax,ax2,ax3,ax4],'fontsize',fs-3)

fprintf('Van der Pol Classical Runge-Kutta Adaptive Step Size μ=3\n')
T = table(reltol, stat.nfun, stat.nAccept, stat.nReject, 'VariableNames',
    ...
    {'Tolerance','N. Func','N. Accept','N. Reject'});
disp(T)
fprintf('Van der Pol Classical Runge-Kutta Adaptive Step Size μ=20\n')
T = table(reltol, stat2.nfun, stat2.nAccept, stat2.nReject, 'VariableNames',
    ...
    {'Tolerance','N. Func','N. Accept','N. Reject'});
disp(T)


%% Exercise 5.5  Compare using different tolerances and step sizes
% with ODE solvers from Matlab
close all; clear all; clc;

fs = 22;
plot_results = true; % Change to not plot figures
mu = [3,20];
x0 = [1.0;1.0];
h0 = 0.001;
tspan = [0 50];

abstol = [1e-3, 1e-7,1e-12];
reltol = abstol;

n = length(abstol);
```

```
% Initialize stat variables
acc3 = zeros(n,1); acc20 = zeros(n,1);
rej3 = zeros(n,1); rej20 = zeros(n,1);
nfun3 = zeros(n,1); nfun20 = zeros(n,1);

% Add fixed RK4 to compare
N = 5000;
[Tf,Xf]=ClassicalRungeKuttaSolver(@VanderPolfunjac,tspan,x0,tspan(2)/N,mu(1)
    );
[T2f,X2f]=ClassicalRungeKuttaSolver(@VanderPolfunjac,tspan,x0,tspan(2)/N,mu
    (2));

for i = 1:n

    % Matlab's ode solvers
    options = odeset("RelTol",reltol(i),"AbsTol",abstol(i));%,'Stats','on');
    [Tmu3,Xmu3, statODE45]=ode45(@VanderPolfunjac,tspan,x0,options,mu(1));
    [Tmu20,Xmu20, statODE45_2]=ode45(@VanderPolfunjac,tspan,x0,options,mu(2)
        );
    [T2mu20,X2mu20,statODE15s]=ode15s(@VanderPolfunjac,tspan,x0,options,mu
        (2));
    [T2mu3,X2mu3,statODE15s_2]=ode15s(@VanderPolfunjac,tspan,x0,options,mu
        (1));

    % Using same initial step size as above for DOPRI54
    [T,X,stat]=ClassicalRungeKuttaAdaptiveStep(@VanderPolfunjac,tspan,x0,h0,
        abstol(i),reltol(i),mu(1));
    [T2,X2,stat2]=ClassicalRungeKuttaAdaptiveStep(@VanderPolfunjac,tspan,x0,
        h0,abstol(i),reltol(i),mu(2));

    if plot_results

        figure, ax=subplot(2,2,1);
        plot(Tmu3,Xmu3(:,1),'r',T,X(:,1),'g',T2mu3,X2mu3(:,1),'m--','
            LineWidth',1),
        hold on,
        plot(Tf,Xf(:,1),'b:','LineWidth',1)
        ylabel('x_1(t)','fontsize',fs),xlabel('t','fontsize',fs),
        xlim([0 50])

        ax2=subplot(2,2,2);
        plot(Tmu3,Xmu3(:,2),'r',T,X(:,2),'g',T2mu3,X2mu3(:,2),'m--','
            LineWidth',1),
        hold on,
        plot(Tf,Xf(:,2),'b:','LineWidth',1)
        ylabel('x_2(t)','fontsize',fs),xlabel('t','fontsize',fs),
        xlim([0,50])
        ax3=subplot(2,2,3:4);
        plot(Xmu3(:,1),Xmu3(:,2),'r',X(:,1),X(:,2),'g',X2mu3(:,1),X2mu3(:,2)
            ,'m--',...
            'LineWidth',1), hold on
        plot(Xf(:,1),Xf(:,2),'b:','LineWidth',1),
        xlim([-4 4]),ylim([-6 6])
        xlabel('x_1(t)','fontsize',fs), ylabel('x_2(t)','fontsize',fs)
        legend('ode45','RK4 (Adaptive)', 'ode15s',...
```

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

```
T=table(reltol',statode45(3,:)',statode45(1,:)',statode45(2,:)', '
    VariableNames', varnames);
disp(T)
fprintf('Van der Pol ODE45 μ=20\n')
T=table(reltol',statode45_2(3,:)', statode45_2(1,:)',statode45_2(2,:)', '
    VariableNames', varnames);
disp(T)
fprintf('Van der Pol ODE15s μ=3\n')
T=table(reltol',statode15s(3,:)',statode15s_2(1,:)',statode15s_2(2,:)', '
    VariableNames', varnames);
disp(T)
fprintf('Van der Pol ODE15s μ=20\n')
T=table(reltol',statode15s_2(3,:)',statode15s(1,:)',statode15s(2,:)','
    VariableNames', varnames);
disp(T)
```

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

A.2.6 Problem 6 Driver

```
%% Problem 6: Dormand-Prince 5(4)
%% Test problem for the test equation
close all; clear all; clc;

fs = 27; % fontsize
x0 = 1;
lambda = -1;
tspan = [0,10];
h0 = 0.01;
abstol = 1e-05;
reltol = 1e-05;

testeq = @(t,x,lambda) x.*lambda; % define test equation function

[T,X,E]=DOPRI54AdaptiveStep(testeq,tspan,x0,h0,...
    abstol,reltol,lambda);

% Plot Analytical solution vs. DOPRI54
fig =gcf;
tANA = linspace(0,tspan(2),1001);
fANA = (exp(lambda*tANA)*x0)';
plot(T,X,'m-o','linewidth',1.2)
hold on
plot(tANA, fANA,'r','linewidth',1),
legend('DOPRI54','Analytical solution')
title({'The Test Equation'},'fontsize',fs)
xlim([0 tspan(2)])
xlabel('t','FontSize',fs-3), ylabel('x(t)','FontSize',fs-3)
set(gca,'FontSize',16)
set(gcf,'paperorientation','landscape')

```

```
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;
```

```
beta = -5:0.01:5;

nreal = length(alpha);
nimag = length(beta);


s = 7; % nr of stages in DOPRI54 method

% Butcher tableau parameters for DOPRI 54
A = zeros(s,s);
        A(2,1) = 1/5;
        A(3,1) = 3/40;
        A(4,1) = 44/45;
        A(5,1) = 19372/6561;
        A(6,1) = 9017/3168;
        A(7,1) = 35/384;
        A(3,2) = 9/40;
        A(4,2) = -56/15;
        A(5,2) = -25360/2187;
        A(6,2) = -355/33;
        A(4,3) = 32/9;
        A(5,3) = 64448/6561;
        A(6,3) = 46732/5247;
        A(7,3) = 500/1113;
        A(5,4) = -212/729;
        A(6,4) = 49/176;
        A(7,4) = 125/192;
        A(6,5) = -5103/18656;
        A(7,5) = -2187/6784;
        A(7,6) = 11/84;
        b = [35/384; 0; 500/1113; 125/192; -2187/6784; 11/84; 0];
        c = [0; 1/5; 3/10; 4/5; 8/9; 1; 1];
        d = [71/57600; 0; -71/16695; 71/1920; -17253/339200; 22/525; -1/40];

I = eye(size(A));
e = ones(size(A,1),1);

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

        absR(kimag,kreal) = abs(R);

    end
end

% Visualize results
figure
fs = 25;
imagesc(alpha,beta,absR,[0 1]); % Solution estimate
grid on
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
        is: %f \n\n',max_dopri)
158
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     abstol,reltol,mu(1));
185
186 [Tout2_VP,Xout2_VP,E2,stat2]=DOPRI54AdaptiveStep(@VanderPolfunjac,tspan,x0,
        h0,...
187     abstol,reltol,mu(2));
188
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: μ=3'},'fontsize',fs-3,'fontweight','bold')
```

```
ylabel('x_1(t)','fontsize',fs-3),xlabel('t','fontsize',fs-3),xlim(tspan)
ax2=subplot(3,1,2);
plot(Tout_VP,Xout_VP(:,2),'r','LineWidth',1.5), %title('\mu = 3'),
ylabel('x_2(t)','fontsize',fs-3),xlabel('t','fontsize',fs-3), xlim(tspan)
ax3=subplot(3,1,3);
plot(Xout_VP(:,1), Xout_VP(:,2),'r','LineWidth',1.5),
xlabel('x_1(t)','fontsize',fs-3),ylabel('x_2(t)','fontsize',fs-3),
ylim([-6 6])
sgtitle({'DOPRI54 - Adaptive Step'},'FontSize', fs)
set([ax,ax2,ax3],'fontsize',fs-4)


figure,
ax=subplot(3,1,1);
plot(Tout2_VP,Xout2_VP(:,1),'r','LineWidth',1.5), %title('\mu = 20'),
subtitle({'Van Der Pol: μ=20'},'fontsize',fs-3,'fontweight','bold')
ylabel('x_1(t)','fontsize',fs-3), xlabel('t','fontsize',fs-3), xlim(tspan)
ax2=subplot(3,1,2);
plot(Tout2_VP,Xout2_VP(:,2),'r','LineWidth',1.5), %title('\mu = 20'),
ylabel('x_2(t)','fontsize',fs-3), xlabel('t','fontsize',fs-3), xlim(tspan)
ax3=subplot(3,1,3);
plot(Xout2_VP(:,1), Xout2_VP(:,2),'r','LineWidth',1.5),
xlabel('x_1(t)','fontsize',fs-3),ylabel('x_2(t)','fontsize',fs-3)
sgtitle({'DORPRI54 - Adaptive Step'},'FontSize', fs)

set([ax,ax2,ax3],'fontsize',fs-4)

% -------- Error estimation and step sizes --------------
figure;f5=gcf;
ax=subplot(2,1,1);
plot(linspace(0,50, length(stat.r)), stat.r, 'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('r','fontsize',fs), %ylim([-0.1 1])
subtitle({'Van Der Pol: μ=3'},'fontsize',fs-3,'fontweight','bold')
ax2=subplot(2,1,2);
plot(linspace(0,50, length(stat.h)), stat.h, 'r','LineWidth', 1),
xlabel('t'),ylabel('h'),
sgtitle('DOPRI54 - Adaptive Step','fontsize',fs+4)

figure; f6=gcf;
ax3=subplot(2,1,1);
plot(linspace(0,50, length(stat2.r)), stat2.r,'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('r','fontsize',fs-2), %ylim([0 1])
subtitle({'Van Der Pol: μ=20'},'fontsize',fs-3,'fontweight','bold')
ax4=subplot(2,1,2);
plot(linspace(0,50, length(stat2.h)), stat2.h, 'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('h','fontsize',fs), %ylim([-0.01 0.3])%
sgtitle('DOPRI54 - Adaptive Step','fontsize',fs+4)

set([ax,ax2,ax3,ax4],'fontsize',fs-3)

fprintf('Van der Pol Explicit Euler Adaptive Step Size μ=3\n')
T = table(reltol, stat.nfun, stat.nAccept, stat.nReject, 'VariableNames',
    ...
    {'Tolerance','N. Func','N. Accept','N. Reject'});
disp(T)
```

```
fprintf('Van der Pol Explicit Euler Adaptive Step Size μ=20\n')
T = table(reltol, stat2.nfun, stat2.nAccept, stat2.nReject, 'VariableNames',
    ...
    {'Tolerance','N. Func','N. Accept','N. Reject'});
disp(T)


%% Test your algorithms on the adiabatic CSTR problem described in the
    papers
% on learn (3D-version and 1-D version)
fs = 24;
t = [0 35];
T0 = 273.65; % in Kelvin for 1D CSTR
x0 = [0,0,T0]'; % for 3D
h0 =0.1;

% Discrete function for flow rate for different times

reltol = 1e-5;
abstol = reltol;

[T,X,E,stat] = DOPRI54AdaptiveStep(@CSTR_1D,t,T0,h0,reltol,abstol,@FlowRate)
    ;
[T3D,X3D,E3D,stat3D] = DOPRI54AdaptiveStep(@CSTR_3D,t,x0,h0,reltol,abstol,
    @FlowRate);

% Get the flow rate trajectory for the plot
count = 0;
for t = 0:0.01:35
    count = count +1;
    F(count) = FlowRate(t);
end

figure
sp=subplot(2,1,1);
ax=plot(T,X-273.15, 'g','LineWidth',1.2);ylabel('T [°C]','fontsize',fs-4);
hold on
ax2=plot(T3D, X3D(:,3)-273.15,'r-.','LineWidth',1.2);
legend([ax,ax2],'1D Version','3D Version','Location', 'NorthWest','fontsize'
    ,13)
sp2=subplot(2,1,2);
plot(0:0.01:35, F,'r', 'LineWidth',1.2),
xlabel('t [min]','fontsize',fs-4),ylabel('F [mL/min]','fontsize',fs-4)
ylim([0 1000])
sgtitle('DOPRI54 - Adiabatic CSTR Simulation','fontsize',fs-1)
set([sp,sp2],'fontsize',15)
set(gcf,'PaperOrientation','landscape')


%% Compare results with ode45 which implements DOPRI54 method and ode15s
% Comparing with the Van der Pol problem stiff vs non-stiff etc.
close all; clear all; clc;

fs = 20;
plot_results = true;
```

```
mu = [3,20];
x0 = [1.0;1.0];
h0 = 0.001;
tspan = [0 50];

abstol = [1e-3, 1e-7, 1e-12];
reltol = abstol;

n = length(abstol);
% Initialize stat variables
acc3 = zeros(n,1); acc20 = zeros(n,1);
rej3 = zeros(n,1); rej20 = zeros(n,1);
nfun3 = zeros(n,1); nfun20 = zeros(n,1);

for i = 1:n

    % Matlab's ode solvers
    options = odeset("RelTol",reltol(i),"AbsTol",abstol(i));%,'Stats','on');
    [Tmu3,Xmu3, statODE45]=ode45(@VanderPolfunjac,tspan,x0,options,mu(1));
    [Tmu20,Xmu20, statODE45_2]=ode45(@VanderPolfunjac,tspan,x0,options,mu(2)
        );
    [T2mu20,X2mu20,statODE15s_2]=ode15s(@VanderPolfunjac,tspan,x0,options,mu
        (2));
    [T2mu3,X2mu3,statODE15s]=ode15s(@VanderPolfunjac,tspan,x0,options,mu(1))
        ;

    % Using same initial step size as above for DOPRI54
    [T,X,E,stat] = DOPRI54AdaptiveStep(@VanderPolfunjac,tspan,x0,h0,abstol(i
        ),reltol(i),mu(1));
    [T2,X2,E2,stat2] = DOPRI54AdaptiveStep(@VanderPolfunjac,tspan,x0,h0,
        abstol(i),reltol(i),mu(2));

    if plot_results

        figure,
        ax=subplot(2,2,1);
        plot(Tmu3,Xmu3(:,1),'r',T,X(:,1),'g',T2mu3,X2mu3(:,1),'m--','
            LineWidth',1),
        ylabel('x_1(t)','fontsize',fs),xlabel('t','fontsize',fs),
        xlim([0 50])

        ax2=subplot(2,2,2);
        plot(Tmu3,Xmu3(:,2),'r',T,X(:,2),'g',T2mu3,X2mu3(:,2),'m--','
            LineWidth',1),
        ylabel('x_2(t)','fontsize',fs),xlabel('t','fontsize',fs),
        xlim([0,50])

        ax3=subplot(2,2,3:4);
        plot(Xmu3(:,1),Xmu3(:,2),'r',X(:,1),X(:,2),'g',X2mu3(:,1),X2mu3(:,2)
            ,'m--',...
            'LineWidth',1),
        xlabel('x_1(t)','fontsize',fs), ylabel('x_2(t)','fontsize',fs)
        xlim([-3 3]),ylim([-6 6])
        legend('ode45','DOPRI54 (Adaptive)', 'ode15s',...
            'Location','SouthEast','fontsize',12)
```

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

```
fprintf('Van der Pol ODE15s μ=3\n')
T=table(reltol',statode15s(3,:)',statode15s(1,:)',statode15s(2,:)', '
    VariableNames', varnames);
disp(T)
fprintf('Van der Pol ODE15s μ=20\n')
T=table(reltol',statode15s_2(3,:)',statode15s_2(1,:)',statode15s_2(2,:)','
    VariableNames', varnames);
disp(T)
```

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

A.2.7 Problem 7 Driver

```
%% Problem 7: ESDIRK23
%% Exercise 7.2 Plot stability region of the ESDIRK23 method
close all; clear all; clc;
mu=[3,20];
x0=[1,1]';
tspan=[0 30];
h0=0.01;

absTol = 10^-6;
relTol = absTol;

alpha = -20:0.01:20;
beta = -20:0.01:20;

nreal = length(alpha);
nimag = length(beta);


gamma = (2-sqrt(2))/2;

a21 = gamma;
b2 = (1-gamma)/2;
b1 = (1-gamma)/2;

for kreal = 1:nreal
    for kimag = 1:nimag
        z = alpha(kreal) + i*beta(kimag);
        R = (1+(b1+b2-gamma)*z + (a21*b2-b1*gamma)*z^2)/((1-gamma*z)^2);
        absR(kimag,kreal) = abs(R);
    end
end

figure
fs = 20;
imagesc(alpha,beta,absR,[0 1]);
grid on
colorbar
axis image
```

```
axis xy
xlabel('Re(z)', 'fontsize',fs-4)
ylabel('Im(z)', 'fontsize',fs-4)
title('|R(z)|', 'fontsize',fs),
set(gca,'fontsize',15)
set(gcf,'PaperOrientation','landscape')

% Check for A stability
lis = 1:nreal;
ind_ltzero=lis(alpha<0);

esdirk_Astable = absR(:,ind_ltzero);
max_esdirk = max(max(esdirk_Astable));

fprintf('The highest value for Re(z)<0 in the stability region for DOPRI54
    is: %f \n\n',max_esdirk)

%  Is the method L stable?
%Lets check some low values...
z= -10000-i*10000;
tmp = (1+b1+b2-gamma);
tmp2 = (a21*b2-b1*gamma)*z^2;
R_esdirk = (tmp+tmp2)/((1-gamma*z)^2);
absR_esdirk = abs(R_esdirk);

fprintf('The stability region for ESDIRK23 when z->inf is: %f \n',
    absR_esdirk)

%% Exercise 7.4 Test algorithm on the Van der Pol problem
close all;clear all;clc

fs = 25;
mu=[3,20];
x0=[1,1]';
tspan=[0 50];
h0=0.001;

absTol = 10^-5;
relTol = absTol;

[Tout_VP,Xout_VP,Gout,info,stat] = ESDIRK23AdaptiveStep(@VanderPolFun,
    @VanderPolJac,tspan,x0,h0,absTol,relTol,mu(1));
[Tout2_VP,Xout2_VP,Gout2,info2,stat2] = ESDIRK23AdaptiveStep(@VanderPolFun,
    @VanderPolJac,tspan,x0,h0,absTol,relTol,mu(2));


% Plot results
figure,
ax=subplot(3,1,1);
plot(Tout_VP,Xout_VP(:,1),'r','LineWidth',1.5), %title('\mu = 3'),
subtitle({'Van Der Pol: μ=3'},'fontsize',fs-3,'fontweight','bold')
ylabel('x_1(t)','fontsize',fs-3),xlabel('t','fontsize',fs-3),xlim(tspan)
ax2=subplot(3,1,2);
plot(Tout_VP,Xout_VP(:,2),'r','LineWidth',1.5), %title('\mu = 3'),
ylabel('x_2(t)','fontsize',fs-3),xlabel('t','fontsize',fs-3), xlim(tspan)
```

```
ax3=subplot(3,1,3);
plot(Xout_VP(:,1), Xout_VP(:,2),'r','LineWidth',1.5),
xlabel('x_1(t)','fontsize',fs-3),ylabel('x_2(t)','fontsize',fs-3),
ylim([-6 6])
sgtitle({'ESDIRK23 - Adaptive Step'},'FontSize', fs)
set([ax,ax2,ax3],'fontsize',fs-4)

% Save figure
%print('VanderPolESDIRK23Adaptive_mu3', '-dpdf','-fillpage')


figure,
ax=subplot(3,1,1);
plot(Tout2_VP,Xout2_VP(:,1),'r','LineWidth',1.5), %title('\mu = 20'),
subtitle({'Van Der Pol: μ=20'},'fontsize',fs-3,'fontweight','bold')
ylabel('x_1(t)','fontsize',fs-3), xlabel('t','fontsize',fs-3), xlim(tspan)
ax2=subplot(3,1,2);
plot(Tout2_VP,Xout2_VP(:,2),'r','LineWidth',1.5), %title('\mu = 20'),
ylabel('x_2(t)','fontsize',fs-3), xlabel('t','fontsize',fs-3), xlim(tspan)
%sgtitle({'Van der Pol Problem', 'ESDIRK23 with Adaptive Time Step'})
ax3=subplot(3,1,3);
plot(Xout2_VP(:,1), Xout2_VP(:,2),'r','LineWidth',1.5),
xlim([-3 3])
xlabel('x_1(t)','fontsize',fs-3),ylabel('x_2(t)','fontsize',fs-3)
sgtitle({'ESDIRK23 - Adaptive Step'},'FontSize', fs)
set([ax,ax2,ax3],'fontsize',fs-4)


% Error estimation and step sizes


% Look at error estimation and step sizes
figure;f5=gcf;
ax=subplot(2,1,1);
plot(linspace(0,50, length(stat.r)), stat.r, 'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('r','fontsize',fs), %ylim([-0.1 1])
subtitle({'Van Der Pol: μ=3'},'fontsize',fs-3,'fontweight','bold')
ax2=subplot(2,1,2);
plot(linspace(0,50, length(stat.h)), stat.h, 'r','LineWidth', 1),
xlabel('t'),ylabel('h'),
sgtitle('ESDIRK23 - Adaptive Step','fontsize',fs)

figure; f6=gcf;
ax3=subplot(2,1,1);
plot(linspace(0,50, length(stat2.r)), stat2.r,'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('r','fontsize',fs-2), %ylim([0 1])
subtitle({'Van Der Pol: μ=20'},'fontsize',fs-3,'fontweight','bold')
ax4=subplot(2,1,2);
plot(linspace(0,50, length(stat2.h)), stat2.h, 'r','LineWidth', 1),
xlabel('t','fontsize',fs),ylabel('h','fontsize',fs), %ylim([-0.01 0.3])%
sgtitle('ESDIRK23 - Adaptive Step','fontsize',fs)

set([ax,ax2,ax3,ax4],'fontsize',fs-6)


```

```
fprintf('Van der Pol ESDIRK23 Adaptive Step Size μ=3\n')
T = table(relTol, info.nFun, info.nAccept, info.nFail, 'VariableNames', ...
    {'Tolerance','N. Func','N. Accept','N. Reject'});
disp(T)
fprintf('Van der Pol ESDIRK23 Adaptive Step Size μ=20\n')
T = table(relTol, info2.nFun, info2.nAccept, info2.nFail, 'VariableNames',
    ...
    {'Tolerance','N. Func','N. Accept','N. Reject'});
disp(T)


%% Exercise 7.5: Compare with Matlabs ODE solvers and other solvers
    developed in
% this exam problem
close all; clear all; clc;

fs = 20;
plot_results = true;
mu = [3,20];
x0 = [1.0;1.0];
h0 = 0.001;
tspan = [0 50];

abstol = [1e-3, 1e-7];
reltol = abstol;

n = length(abstol);
% Initialize stat variables
acc3 = zeros(n,1); acc20 = zeros(n,1);
rej3 = zeros(n,1); rej20 = zeros(n,1);
nfun3 = zeros(n,1); nfun20 = zeros(n,1);


for i = 1:n

    % Matlab's ode solvers
    options = odeset("RelTol",reltol(i),"AbsTol",abstol(i));%,'Stats','on');
    [Tmu3,Xmu3, statODE45]=ode45(@VanderPolfunjac,tspan,x0,options,mu(1));
    [Tmu20,Xmu20, statODE45_2]=ode45(@VanderPolfunjac,tspan,x0,options,mu(2)
        );
    [T2mu20,X2mu20,statODE15s_2]=ode15s(@VanderPolfunjac,tspan,x0,options,mu
        (2));
    [T2mu3,X2mu3,statODE15s]=ode15s(@VanderPolfunjac,tspan,x0,options,mu(1))
        ;

    % Using same initial step size as above for ESDIRK23
    [T,X,Gout,info,stat] = ESDIRK23AdaptiveStep(@VanderPolFun,@VanderPolJac,
        tspan,x0,h0,abstol(i),reltol(i),mu(1));
    [T2,X2,Gout2,info2,stat2] = ESDIRK23AdaptiveStep(@VanderPolFun,
        @VanderPolJac,tspan,x0,h0,abstol(i),reltol(i),mu(2));
    if plot_results

        figure,
        ax=subplot(2,2,1);
```

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

```
varnames = {'Tolerance','N. Func','N. Accept','N. Reject'};
fprintf('Van der Pol ESDIRK23 Adaptive Step Size μ=3\n')
T=table(reltol', nfun3, acc3, rej3, 'VariableNames', varnames);
disp(T)
fprintf('Van der Pol ESDIRK23 Adaptive Step Size μ=20\n')
T=table(reltol', nfun20, acc20, rej20, 'VariableNames', varnames);
disp(T)
fprintf('Van der Pol ODE45 μ=3\n')
T=table(reltol',statode45(3,:)',statode45(1,:)',statode45(2,:)', '
    VariableNames', varnames);
disp(T)
fprintf('Van der Pol ODE45 μ=20\n')
T=table(reltol',statode45_2(3,:)', statode45_2(1,:)',statode45_2(2,:)', '
    VariableNames', varnames);
disp(T)
fprintf('Van der Pol ODE15s μ=3\n')
T=table(reltol',statode15s(3,:)',statode15s(1,:)',statode15s(2,:)', '
    VariableNames', varnames);
disp(T)
fprintf('Van der Pol ODE15s μ=20\n')
T=table(reltol',statode15s_2(3,:)',statode15s_2(1,:)',statode15s_2(2,:)','
    VariableNames', varnames);
disp(T)


%% Exercise 7.5 cont. Comparing with the other solvers developed in this
    exam problems
close all; clear all; clc;

fs = 20;
plot_results = true;
mu = [3,20];
x0 = [1.0;1.0];
h0 = 0.001;
tspan = [0 50];

abstol = [1e-3, 1e-7];
reltol = abstol;
n = length(abstol);

% Initialize stat variables
acc3 = zeros(n,1); acc20 = zeros(n,1);
rej3 = zeros(n,1); rej20 = zeros(n,1);
nfun3 = zeros(n,1); nfun20 = zeros(n,1);

for i = 1:n

    % Classical Runge-Kutta
    [Tmu3,Xmu3, statRK4] = ClassicalRungeKuttaAdaptiveStep(@VanderPolfunjac,
        tspan,x0,h0,abstol(i),reltol(i),mu(1));
    [Tmu20,Xmu20, statRK42] = ClassicalRungeKuttaAdaptiveStep(
        @VanderPolfunjac,tspan,x0,h0,abstol(i),reltol(i),mu(2));

    % Explicit Euler
    [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,
            tspan,x0,h0,abstol(i),reltol(i),mu(1));
293     [TIEmu20,XIEmu20, statIE2]= ImplicitEulerAdaptiveStep(@VanderPolfunjac,
            tspan,x0,h0,abstol(i),reltol(i),mu(2));
294
295
296     % Dopri54
297     [TD, XD, ED, statD] = DOPRI54AdaptiveStep(@VanderPolfunjac,tspan,x0,h0,
            abstol(i),reltol(i),mu(1));
298     [TD2, XD2, ED2, statD2] = DOPRI54AdaptiveStep(@VanderPolfunjac,tspan,x0,
            h0,abstol(i),reltol(i),mu(2));
299
300
301     % ESDIRK23
302     [T,X,Gout,info,stat] = ESDIRK23AdaptiveStep(@VanderPolFun,@VanderPolJac,
            tspan,x0,h0,abstol(i),reltol(i),mu(1));
303     [T2,X2,Gout,info2,stat2] = ESDIRK23AdaptiveStep(@VanderPolFun,
            @VanderPolJac,tspan,x0,h0,abstol(i),reltol(i),mu(2));
304
305     if plot_results
306
307         figure,
308         ax=subplot(2,2,1);
309         plot(Tmu3,Xmu3(:,1),'r',T,X(:,1),'g',T2mu3,X2mu3(:,1),'m--','
                LineWidth',1),
310         hold on, plot(TIEmu3,XIEmu3(:,1),'b:',TD, XD(:,1),'k','LineWidth',1)
311         ylabel('x_1(t)','fontsize',fs),xlabel('t','fontsize',fs),
312         xlim([0 50])
313
314         ax2=subplot(2,2,2);
315         plot(Tmu3,Xmu3(:,2),'r',T,X(:,2),'g',T2mu3,X2mu3(:,2),'m--','
                LineWidth',1),
316         hold on, plot(TIEmu3,XIEmu3(:,2),'b:',TD, XD(:,2),'k','LineWidth',1)
317         ylabel('x_2(t)','fontsize',fs),xlabel('t','fontsize',fs),
318         xlim([0,50])
319
320         ax3=subplot(2,2,3:4);
321         plot(Xmu3(:,1),Xmu3(:,2),'r',X(:,1),X(:,2),'g',X2mu3(:,1),X2mu3(:,2)
                ,'m--',...
322             'LineWidth',1), hold on,
323         plot(XIEmu3(:,1),XIEmu3(:,2),'b:',XD(:,1), XD(:,2),'k','LineWidth'
                ,1)
324         xlabel('x_1(t)','fontsize',fs), ylabel('x_2(t)','fontsize',fs)
325         xlim([-3 3]),ylim([-6 6])
326         legend('RK4','ESDIRK23', 'ExplicitEuler','ImplicitEuler','DOPRI54'
                ,...
327             'Location','northwest','fontsize',12.7)
328         sgtitle({'Van der Pol: $\mu = 3$'},'fontsize',fs+5),
329         set([ax ax2 ax3],'fontsize',fs-3)
```

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

```
370             [statRK42.nReject,statEE2.nReject,statIE2.nReject,statD2.nReject,
                    info2.nFail]', 'VariableNames', varnames);
371     disp(Table2)
372
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=IwAR30x3DslEucGoIfdRJflsJTWTllVxI5B_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.